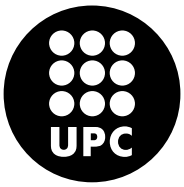


UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Evaluation of Low-Power Architectures in a Scientific Computing Environment

Author:
Constantino Gómez Crespo

MSc Innovation and Research in Informatics
High Performance computing and Computer Architecture

7th of July, 2016

Advisor: Eduard Ayguadé, DAC - UPC
Co-advisor: Filippo Mantovani, Barcelona Supercomputing Center

Contents

List of Figures

List of Tables

Summary

HPC (High Performance Computing) represents, together with theory and experiments, the third pillar of science. Through HPC, scientists can simulate phenomena otherwise impossible to study. The need of performing larger and more accurate simulations requires to HPC to improve every day.

HPC is constantly looking for new computational platforms that can improve cost and power efficiency. The Mont-Blanc project is a EU funded research project that targets to study new hardware and software solutions that can improve efficiency of HPC systems. The vision of the project is to leverage the fast growing market of mobile devices to develop the next generation supercomputers.

In this work we contribute to the objectives of the Mont-Blanc project by evaluating performance of production scientific applications on innovative low power architectures. In order to do so, we describe our experiences porting and evaluating sate of the art scientific applications on the Mont-Blanc prototype, the first HPC system built with commodity low power embedded technology. We then extend our study to compare off-the-shelves ARMv8 platforms. We finally discuss the most impacting issues encountered during the development of the Mont-Blanc prototype system.

Chapter 1

Introduction

Nowadays, High Performance Computing is used as a tool that allows the research community to push science forward. In the same way, it has also a positive impact in industry and our society. The need of performing larger and more accurate simulations requires to HPC to improve every day. Current research in HPC is trying to tackle many of the challenges that affect the development of the next generation of supercomputers.

Historically in HPC, special purpose technology has been always replaced by a more cheaper and competitive commodity technology available in the market. For example, although vector processors were especifically designed for HPC environments, RISC and later x86 processor architectures replaced them as a dominating HPC systems technology[25]. Given the current massive mobile market, it is possible that in a similar way, mobile architectures become the next commodity technology that replaces x86.

This chapter introduces first the production scientific applications and parallelism concepts. In the background section we present the current state of the art and challenges in the topics related to the area of HPC architecture and scientific applications. Later, we talk about the motivation and main objectives of our work and finally, we discuss similar projects and research in the related work subsection. The

rest of the thesis is structured as follows: we introduce the architecture details of the computing clusters involved in our study in chapter 2, in chapter 3 we describe the set of applications and benchmarks we used to evaluate and compare those platforms. We present the results of our experiments and issues in chapters 4 and 5. Finally, we share our conclusions in chapter 6.

1.1 Production Scientific Applications

We refer to *production* applications to those scientific applications that are used as a tool to perform real simulations whose results allow an improvement in the scientific knowledge e.g. simulations generating data for a paper in a given scientific area. We also include in this category the complex codes, usually used in industrial R&D environment for development of complex products e.g. Computer Fluid Dynamics (CFD) software to perform wind tunnel simulations in a car company. As opposite to production, we call *benchmark* or *mini-applications*, the software which purpose is to provide a synthetic framework to test the performance of the applications on different platforms or hardware configurations, or to validate itself as a valid scientific computing method.

Performing real scale simulations in industry and science requires high quantity of compute resources both in terms of CPU time and memory. Regular personal computers and servers can not perform those simulations mainly because of memory limitations or lack of computational power. In these cases, researchers make use of HPC clusters, also called Supercomputers. Supercomputers are built using a high number of computational nodes, each node with one or more processors, interconnected with a high-speed network. Scientific applications (can) take advantage of this architecture by splitting the computational problem in tasks, and distributing those tasks among the computational nodes. This way scientists can exploit the computational power of several nodes working in parallel and split bigger problems among the memories of the nodes.

Developing applications that can run in parallel is not trivial. Developers must

take design decisions and face a set of technical issues based on the target cluster architecture. Exploiting parallel efficiency at high number of cores requires a deep knowledge about the application algorithms and hardware details. In this thesis, we discuss about this technical issues and challenges of porting and executing scientific applications on a HPC cluster based in low power technology.

1.2 Parallelism and Scalability

Parallel programming is not an emerging paradigm anymore and also not restricted to the HPC world. Paralelism is found and exploited in variety of workloads, from user mobile applications to embedded real time systems in industry, and all the so called *cloud services* in between. To be able to execute applications in parallel we have to find concurrency in them, to do that, we break a problem into discrete parts (tasks) and determine dependencies between them, two tasks that do not depend on each other are concurrent and potentially can be executed in parallel.

Although the previous description refered to tasks in general, we can distinguish three main sources of parallelism based on its granularity.

Task Level Parallelism Also known as Thread level parallelism. Execution in parallel functions and blocks of instructions without data dependeces between them.

Data Level Parallelism The same block of instructions is executed with different data. SIMD instructions and GPU floating-point (FP) units exploit this. Also, it is common in parallel programs to leverage DLP to easily transform the code to be executed in different threads obtaining TLP e.g. a for struct paralelized with *#pragma omp parallel for*.

Instruction Level Parallelism Instructions can be executed in parallel or re-ordered. For example, out-of-order processor architectures leverage the instruction flow paralellism to obtain performance benefits.

To scale applications to a high number of cores, the most important types of parallelism are the task level and data level parallelism, which usually allows the creation of threads of a reasonable size with no dependencies.

1.3 Background and State of the Art

The performance curve along the years show an exponential increase of the peformance of the Supercomputers in the top 500. This trend is expected to continue [31] but not without first tackling many of the challenges of the current commodity HPC cluster technology.

In the last decade, the most common way to build supercomputers is the so called cluster architecture. A HPC cluster is build interconnecting high number of nodes, where each node has the same hardware components. The state of the art hardware found in HPC clusters is based on x86 server-class node technology with high speed fiber optics network interconnects. However, its becoming usual that the most powerful machines in supercomputing also include compute accelerators on its architecture.

The next milestone in HPC will be achieving at least one exaflop compute capacity. In order to build the next generation of supercomputers, the HPC research community must tackle one by one, several important issues found in the current technology. In the latest years, supercomputers have been designed with several constraints in mind that have become progressively outdated. For example, the primary technological constraint to build HPC systems is not longer the peak clock frequency, instead, we are now limited by power consumption. Floating point compute capacity relevance has been displaced by the need of minimizing the data movement in the system. Exascale machines will require applications that exploit parallelism up to millions of cores and develop new techniques to reduce the performance gap withi CMPs (Chip Multi-processors) between the memory and the cores[27]. This breakthrough in compute performance will directly benefit many state of the art applications allowing the research community to perform more advanced simulations.

For example in combustion research, it will allow to move from simulations using simplified models of rocket engines to full detailed models of those parts[27].

1.3.1 Multicore Performance and the Memory Wall

Memory bandwidth is a critical resource in multicore systems. The processor performance is doubling every two years while DRAM performance is doubling every three years; every year the performance gap between both is increased. This trend, which is expected to continue, is also known as *the memory wall*.

DRAM has been used for years to build memory DIMMs because it is very convinient; fast, simple and cheap to manufacture. However, scaling of DRAM technology is expected to end in the next years[21]. Several alternatives have been proposed to improve or replace DRAM, for example: TSV (Through-Silicon Vias) 3d stacked memory aims to improve memory performance by heavily increasing the memory bandwidth from the processor to the memory banks; and non-volatile emerging technologies which aim to replace the current DRAM cell technology with more energy efficient, non-volatile memory cells with similar speed to DRAM and better scaling potential [34]. 3d stacked memory is used in the latest generation of Intel Xeon Phi accelerators and non-volatile technology in main memory will be included in the architecture of new supercomputers [19].

1.3.2 Level of Parallelism in production applications

As we mention, developing production application that exploit efficiently the compute capabilities of a HPC cluster is not trivial. Our maximum level of parallelism is determined by the size of the total parallel region of an application [37]; that is in this case, the region that exhibits task level or data level parallelism. In HPC we have to deal with several sources of inefficiency or overheads that can be classified in MPI transfers, serialization and load unbalance.

Although there is plenty of techniques to hide or solve the most common sources of overheads, it is also common to find production applications that does not apply them. For example, even if there are several libraries that allow us to write files to disk in parallel, we often find applications that still use only one thread to do it causing the parallel efficiency to drop very low when scaling to high number of cores due to serialization. Note that, not all optimizations might be suitable for an application, or the difficulty to apply them in the code might be very high, therefore it is necessary to study in each case which optimizations would yield higher parallelism improvements.

1.3.3 Network interconnections

In distributed memory systems, we rely on off-chip message passing over the network to send and recieve data from remote nodes. Some applications require collective all-to-all comunications, meaning that every node has to send and recieve information from all of the other nodes. Dealing with such collective operations are a big challenge for HPC systems with more than thousands of nodes; network topologies with good collective performance might be less powerful in other communication patterns. 3D torus interconnects are designed to exploit better communications with neighboring nodes while a standard fat tree interconnect provides cost efficient interconnection with a lower maximum number of hops between nodes. For example, BlueGene/L included [3] separate interconnection hardware for MPI collective operations and the rest of MPI operations. The design of new topologies will be key to enable high parallel efficiency in next generation exascale supercomputers.

1.3.4 Energy Efficiency

Energy consumption is one of the main constraints to build the next generation of supercomputers. If we take a look at the green500 list we observe that the top of the list is populated with systems using compute accelerators. In this case, such accelerators pack a high density of floating point units connected to high bandwidth memory. The price to pay for this performance at low energy cost is the use of

a dedicated programming interface. Although elaborate implementations of HPL can take advantage of the compute power of those accelerators, exploiting them its not always possible, or at least not an easy task, in other kinds of applications. Therefore, to really develop an efficient machine running production applications we can not only rely on accelerators, we must take in consideration other parts of the cluster achitecture in order to save energy, for example: processor, memory, network and cooling system technology. Processor power dissipation is mostly determined by three factors, feature size, frequency and microarchitectural components like the size of the caches; the impact of the ISA being RISC or CISC in nowadays processors is negligible[5]. We mentioned already, that new emerging technologies are set to take over and replace DRAM in future exascale systems [34]. As for cooling systems, liquid submersion featured by the first time in TSUBAME KFC [10] in a large scale supercomputer is considered of the most energy efficient approaches to cool a system, however, we have to consider other aspects like the scalability and cost of such technology. Finally, we also have to keep in mind that developing parallel efficient applications also improves performance per watt by not wasting resources during the execution of an application.

1.3.5 Mobile low power architectures for HPC

To justify the approach of using mobile technology to build HPC systems it is necessary to look at the evolution of performance of mobile SoCs in the latest years compared to the x86 processors used in HPC.

In figure 1.1 we observe how the performance gap of mobile SoCs against current comodity HPC technology is closing rapidly. The main reason for this rapid increment in performance is the current market volume supporting using this technology. This phenomenon opens the door to another shift, this time from x86 to ARM architecture in HPC processors due to its relative low cost.

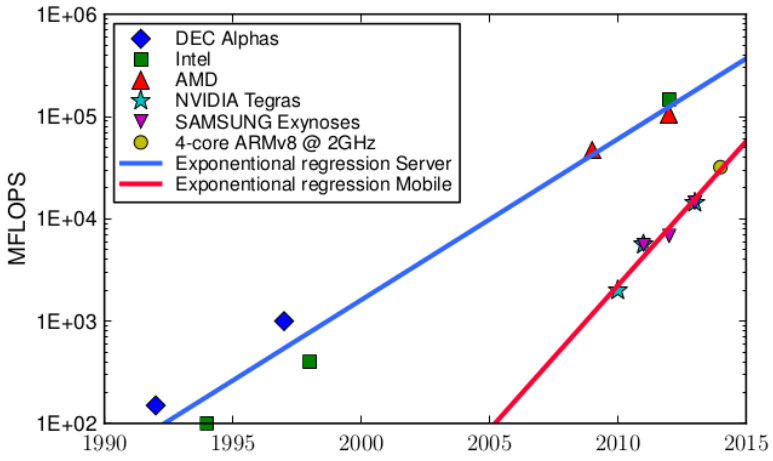


Figure 1.1: Evolution of performance of mobile vs hpc commodity processors.[25]

1.4 Motivation

The work presented in this thesis has been performed within the BSC (Barcelona Supercomputing Center) in the context of the national Severo Ochoa programme and the EU funded Mont-Blanc project. In this section, we describe the main motivation and goals of these projects, highlighting the original contributions of this thesis to those goals.

In 2011, and renewed in 2015, BSC receives the Severo Ochoa Center of Excelence accreditation from the Spanish Government, acknowledging the important contributions of the supercomputing resources to the scientific community, while at the same time, develops its own research lines (Computer sciences, Earth Sciences, Life Sciences and Engineering Applications).

In order to keep promoting the cross department collaboration that benefits both ends, the center initiate several projects and partnerships, one of those being the study of *feasibility and performance of the scientific applications in low power architectures*, which brings together relevant in-house scientific applications and the new class of energy-efficient architectures being developed within the Mont-blanc project.

The vision of the Mont-Blanc project is to leverage mobile technology for devel-

opment of next generation HPC systems. Various observations hide behind this vision:

1. mobile market can leverage large volumes and consequently lower prices
2. mobile devices are intrinsically sensitive to power efficiency
3. computational power of mobile devices is increasing faster than standard HPC technology

More details about the Mont-Blanc prototype can be found in section 2.2.1.

As we will see, our work contributes to the following specific objectives of the Mont-Blanc project:

- To develop a fully functional energy-efficient HPC prototype using low-power commercially available embedded technology.
- To design a next-generation HPC system together with a range of embedded technologies in order to overcome the limitations identified in the prototype system.
- To complement the effort on the Mont-Blanc system software stack, with emphasis on programmer tools, system resiliency, and ARM 64-bit support.

Tasks and goals

Within this framework, we establish these specific tasks and goals.

1. **Deploy and run succesfully in the Mont-blanc prototype the set of in-house BSC Scientific applications.**
    - Alya RED from CASE department
    - NMMB from Earth Sciences department
    - SMuFiN from Life Sciences department
    - And Saiph (DSL) from Computer Sciences department

This point includes porting of the applications, collaborating with the sysadmin prototype team to prepare the correspoding software stack to support each application, and verification of the results obtained.

2. **Analysis and evaluation of performance.** Obtaining and reporting scaling, parallel efficiency and energy consumption metrics of each application. Analysing the performance gap comparing results obtained on the Mont-Blanc platforms with state of the art cluster technology (Intel Xeon). Trace driven deep analysis for fine-grain identification and quantification of cluster issues.

3. **Optimizations, issue solving and experiences.** Gathering of experiences while solving application and platform issues, applying specific optimizations of the cluster architecture. Evaluation of the impact of those optimizations and fixes.

4. **Extension of the study to available ARMv8 platforms.** Although at the beginning of this work the platforms only 32 bit platforms were available on the market, meanwhile 64 bit ARM based platforms made their appearance on the market. As the Mont-Blanc project acquired two mini-cluster based on this technology (see section 2.2.2 and 2.2.3) it has been possible to perform an evaluation on those platforms as well.

1.5 Related work

Researchers at CERN published studies showing the viability of scientific codes on low power ARM platforms using Odroid-U2 and XGene2 development kits in a single node configuration[2] [1]. Before the Mont-Blanc prototype the Mont-Blanc team assembled smaller prototypes, Tibidabo [26], Pedraforca and a Arndale Board based mini-cluster[25], showing an in depth analysis and evaluation of the market available technology in order to develop a future full size high performance computer based on mobile ARM processors.

Chapter 2

The Montblanc prototype and mini Clusters

In this chapter we describe the architecture of all the platforms used in our work and experiments. First, we describe the architecture of a *generic* state of the art HPC cluster, the MareNostrum3 supercomputer. Second, we describe the Mont-Blanc prototype and mini-clusters, platforms in which we have been active in the development, testing and evaluation.

2.1 General Cluster Description

This section describes the MareNostrum 3 supercomputer at BSC with two purposes: first, provide information about a plaftorm used in our comparisons, and also, give an example of a state of the art supercomputer architecture that is used for scientific computing.

With a peak performance of 1.1 Petaflops, the MareNostrum 3 supercomputer is placed in the 106 position in the top 500 ranking (April 2016 list [32]). It is a Tier-0 supercomputer system of the PRACE network, which provides supercomputing resources access to research centers across Europe.

Currently it has 3056 nodes, a total of 115.5 TB of main memory and 2 PB of disk storage. Compute nodes are interconnected using an Infiniband FDR10 fat tree network topology. Each node has two sockets with Intel Xeon E5-2670 processors, together they achieve a peak performance of 332.8 GFLOP/s. At 8 cores per socket, that makes up to a total of 48896 cores in the whole system. In figure 2.1, we see a detailed schema of a MareNostrum3 compute node. Each node has 32 GB of DDR3 RAM @ 1600 MHz distributed in eight 4GB DIMMs. Both sockets have four memory channels each and perform non-uniform memory accesses and mantain cache coherence between them through QPI. Additionally, each node also has a Mellanox ConnectX-3 network card that interfaces directly with a socket via PCI-Ex8, two ethernet network cards to access the management network and GPFS and a local 500 GB hard drive. The whole system is cooled using mainly water cooling, although there is also small fans placed behind each pair of nodes.

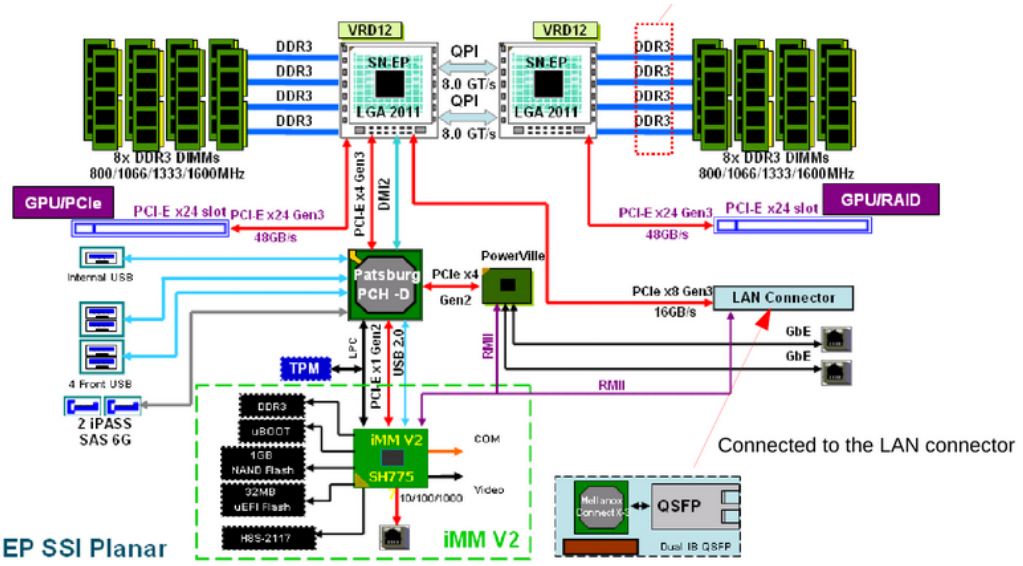


Figure 2.1: MareNostrum 3 Compute node block diagram

It is usual that the architectural features of supercomputers are influenced by the computational requirements of the applications that the machine is intended to run. As we mention in section 1.3.1, memory bandwidth is key to achieve multicore performance, specially in memory intensive applications. Similarly, efficient network interconnections are necessary to scale distributed applications to a high number of

cores.

Other important part of the supercomputer is the software stack. The software stack is composed of all the necessary packages to execute applications, from the operating system to the scientific libraries and compilers. To allow the users to easily compile, install and run applications a cluster it is usual to manage the operating system binary and library paths using a module manager. A module manager allows us to modify userspace environment variables to work with specific versions of software, for example, if we require to compile our software with an older version of a library for compatibility. Also, to manage efficiently the resources in a HPC cluster it is necesary to use a job queue manager. A user that wants to run an application has to write a small bash script that is submmited to a queue, additionally to the execute command, those scripts usually include information about the number of cores required to run the job and how long and in which folder the ouput will be placed. A job queue manager uses that information to schedule in the best possible way the jobs in the cluster. MareNostrum 3 uses a GNU module manager and IBM LSF job queue manager.

Nowadays, it is common that these platforms offer power monitoring inside each node. Fine-grained energy monitoring keeps sampling data extracted from the power monitor and exposes it to the user, allowing him to obtain more detailed power profiles of a job execution. Coarser-grained tools only report a total measurement at the end of a job. MareNostrum 3, reports the total energy consumption (kWh) of all the nodes during a job execution. Additionally to the power monitoring hardware, it is also possible to obtain power measurements from hardware counters (Running Average Power Limit) available at Intel processors.

2.2 Mont-Blanc philosophy and platforms

As we mention before (see section 1.4), the main objetive in the Mont-Blanc project is to contribute to the development of next generation energy-efficient HPC systems. In the project, all designs of new platforms and architectures are developed using

the same philosophy. Decisions about what technology to install are based on the current commodity market. In detail, first we identify the platforms that satisfy this simple characteristics: 1) they can run Linux, 2) they have node interconnection capabilities. Then, we compare them in terms of prices, socket peak performance, network capabilities and cooling; considerations needed to ensure scalability. Once we select a platform we produce a small cluster for testing before building a bigger full scale version of it.

2.2.1 The Mont-Blanc prototype

The Mont-Blanc prototype is the result of the first phase of the Mont-Blanc European project, an international collaboration of 14 academic and industrial partners. The vision of the project is to leverage commodity embedded technology, i.e. the one of smartphone and tablets, for performing scientific computation. The main objective of the first phase of the project, ended in July 2015, was to deploy a full scale prototype based on mobile technology. This objective has been succesfully achieved between January and July 2015 when the Mont-Blanc prototype has been installed at the Barcelona Supercomputing Center.

The Mont-Blanc prototype has a total of 1080 nodes, being 1064 compute nodes and 16 login nodes. They are organized in blades, chassis and racks as follows: each blade contains 15 compute nodes, each chassis contains 9 blades and each rack has 4 chassis, with a total of two racks. That is, 135 nodes per chassis and 540 per rack. Each node has two cores adding a total of 2160 cores in the system, 2128 available to run parallel applications.

Figure 2.2 shows a picture of the prototype at BSC facilities. Both racks are $42U - 19$" standard format.

Figure 2.2: Mont-Blanc prototype racks at Barcelona Supercomputing Center

Compute node

The Mont-Blanc compute node uses a Server-on-Module architecture. Each node card (Samsung Daughter Card or SDB) has one Samsung Exynos 5250 that includes two ARM Cortex-A15 CPU running at 1.7 GHz and an ARM Mali GPU. More specific details about the SoC can be found in table 2.1. Main memory is located on the board connected to the SoC via two memory channels. A μSD slot allows us to connect a 16 GB μSD card which includes the boot-loader, operating system and local storage. Each SDB is connected to the blade board through PCI-e 4x and has a TDP of 15W. The peak DP floating-point performance of each core is 3.4 GFLOPS and the Mali GPU, 21.3 GFLOPS, for a total of 28.1 GFLOPS per compute node. Figure 2.3 shows the physical layout of the components over the SDB.

System-on-Chip		
	CPU	**GPU**
Type	2 x ARM Cortex-A15	ARM Mali T-604
Frequency	1.7 GHz	533 MHz
LLC	1M L2 Shared	-
Peak performance (DP)	6.8 GFLOPS	21.3 GFLOPS
Main Memory		
Type		LPDDR3-1600 RAM
Size		4 GB
Memory channels		2
Shared GPU and CPU		Yes
Peak bandwidth		12.8 GB/s
Network interconnect		
NIC		USB 3.0 to Ethernet bridge
Link		1 Gbps

Table 2.1: Mont-Blanc prototype SDB hardware characteristics.

Network interconnection

With high number of nodes, we require an efficient and scalable topology to achieve good performance in MPI applications. Mont-blanc prototype integrates a fat tree topology on three levels. The first level of switching is implemented inside the blades using a 1 GbE embedded switch with 2x10 GbE up-links. The second level interconnects all blades in a rack with 10 GbE links. At the top, the third level of the tree interconnects both rack switches using 4x40 GbE links.

Software stack

One of the objectives of the Mont-Blanc project is to develop the HPC software ecosystem required to run HPC applications like in any other x86 based cluster. With this we want to help pave the way for market acceptance of ARM solutions. Figure 2.4 shows the software stack available at the prototype.

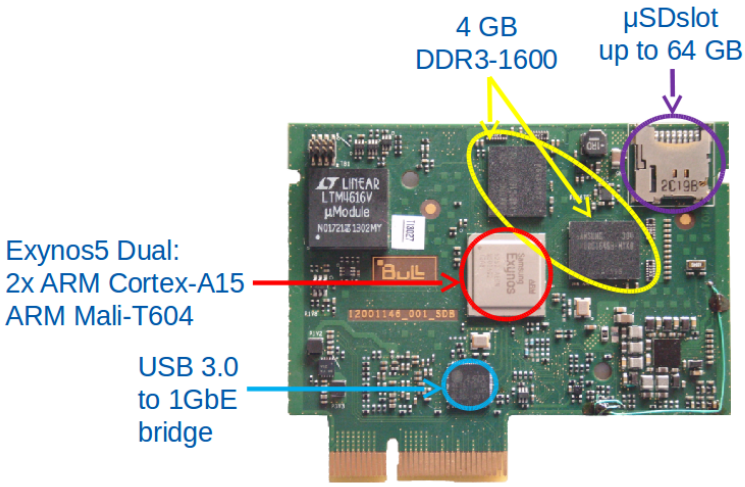


Figure 2.3: Mont-Blanc prototype compute node (SDB)

Other information

In figure 2.5 we show a picture of the Mont-Blanc prototype building blocks, blade (2.5a) and chassis (2.5b). Following, we describe the rest of important aspects of the prototype:

Power monitoring The system collects power samples using sensors connected to the supply rails of each SDB. Every $\approx$ 1100ms a FPGA collects the average value mesured in those sensors and stores them in an intermediate buffer. Then, using the Board Manager Controller (BMC) in the blade board, a system monitoring tool extracts data from such buffer along with a timestamp and stores it in a database; which is accesible to the prototype users.

Storage The storage system is connected to the rack switches through four 10 GbE cables. It uses Lustre as a parallel file system.

Cooling Each blade has its own set of fans in the front (left of the picture), also, each SDB card mounts a heat sink.

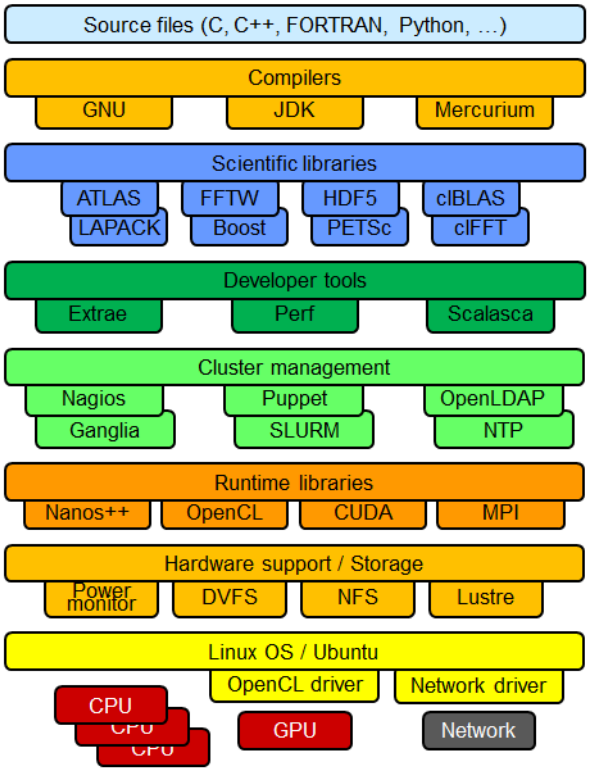


Figure 2.4: Mont-Blanc prototype sotftware stack

2.2.2 XGene2 mini-cluster

Following the *Mont-Blanc philosophy*, this mini-cluster is built with the purpose of evaluating the capabilites of ARMv8 commercially available platforms for HPC. The development related to this platform takes place during the second phase of the Mont-Blanc project in Q1 2016.

For each XGene2 mini-cluster node we use a single socket Applied Micro Merlin server board based on its custom ARMv8 core implementation XGene2. We have a total of four Merlin boards, we use three as compute nodes, that is 24 cores available for running jobs, and the last one as login node. The form-factor of the node is a standard 1U.

APM883408-X2 is the codename of the System-on-Chip that mounts the Merlin

(a) Blade

(b) Chassis

Figure 2.5: Mont-Blanc prototype building blocks.

board. This processor features: eight XGene2 cores running at 2.4 GHz, 256KB Shared L2 per pair of cores with ECC, 8MB Shared L3 protected by parity, four on-die memory controllers also with ECC support and two 10 Gbps Ethernet with link aggregation and RDMA over ethernet support.

Figure 2.6 shows the layout of components and connectos of the Merlin board. Merlin comes with 128 GB of DDR3 ECC in eigh 16 GB DIMMs to take advantage of the four memory controllers. In our cluster configuration we connected all boards using a single switch and one 10 GbE fiber optics link per board in a point-to-point topology. For cooling the components the board includes both heat sinks on top of the processors and 6 fans aligned in the front inside the 1U chassis. The storage is shared across mini-clusters and it uses a remote NFS file system.

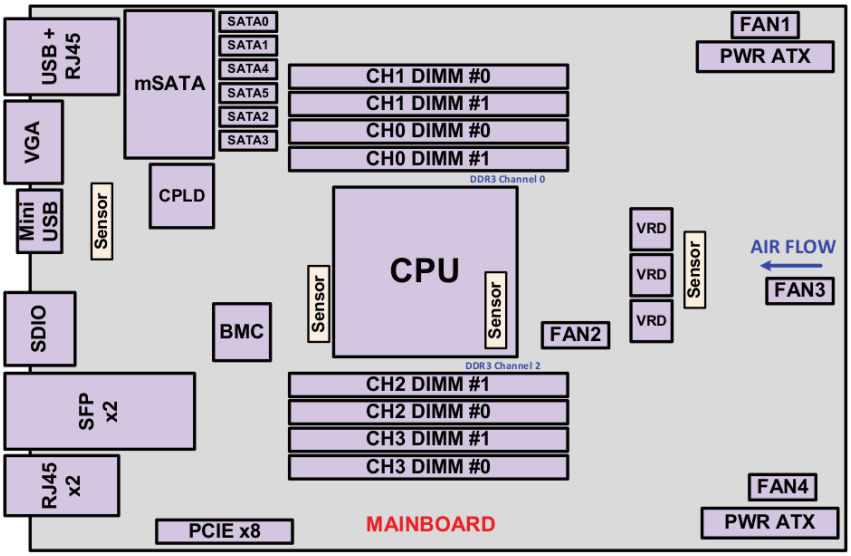


Figure 2.6: Layout of the components and connectors on the Merlin Board

2.2.3 ThunderX mini-cluster

With the same purpose and during the same timeframe as XGene 2 mini-cluster (see 2.2.2), we also deploy mini-cluster using Cavium ThunderX 2K server system as a compute node. We have available 4x Cavium ThunderX 2K boards mounted in a 2U chassis that we use as four independent compute nodes. As a login node we reuse the same node used in the ARMv8 based XGene2 mini-cluster. Each board features 96 Cavium ThunderX cores, therefore we have a total of 384 in the whole cluster.

Figure 2.7 shows a picture of the board containing both sockets, memory slots and additional components. More in detail, the board includes two 48-Core processors (codename CN8890) connected in dual socket cache coherent configuration through CCPI (Cavium Coherent Processor Interconnect). Each processor runs at 1.8 GHz has a 16MB L2 (LLC) shared cache, network controller with support for 2 x 10 GbE and 1 x 40 GbE. Our board configuration includes 128 GB (8 x 16 GB) DDR3 ECC RAM running at 1880 MHz.

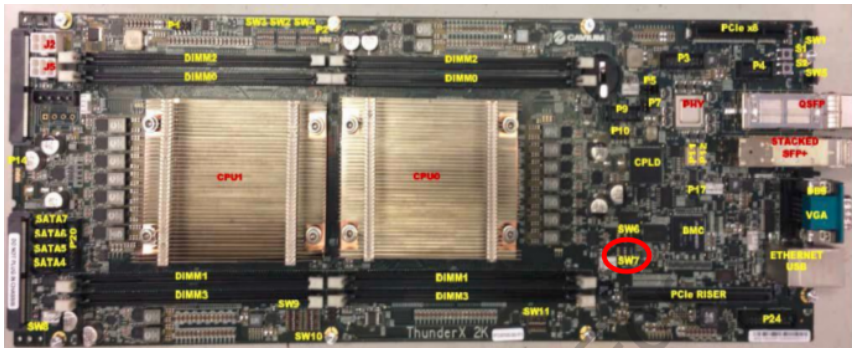


Figure 2.7: Picture of a Cavium ThunderX 2K server board.

Given the low number of nodes, we interconnect them with the same point-to-point topology as in XGene2 cluster, but in this case, using two 10 GbE cables with link agreggation. Each node has 1TB SSD for local storage and is connected to a remote NFS file system.

2.2.4 JetsonTK1 mini-cluster

JetsonTK1 mini-cluster is an ARMv7 platform built using 8 Nvidia Jetson TK1 development boards; 7 compute nodes, 1 login node. Interconnected using 1 GbE links in a point-to-point topology. Each board features: a 4-plus-1 multicore processor (4x ARM Cortex-A15 + 1x Cortex-A7), a Kepler GPU with 192 cores, 2 GB of DDR3 RAM, a single memory controller with 64-bit width shared between the GPU and the CPU, and a 1 Gb dedicated Ethernet port.

2.3 Compute node cost

It is not possible por us to provide a breakdown of the costs of the Mont-Blanc prototype or many of the boards we use to build our mini-clusters. It is important to remark that very often the final price depends in big measure on the negotations with the provider. Also, it is difficult to reason about the cost effectiveness of our

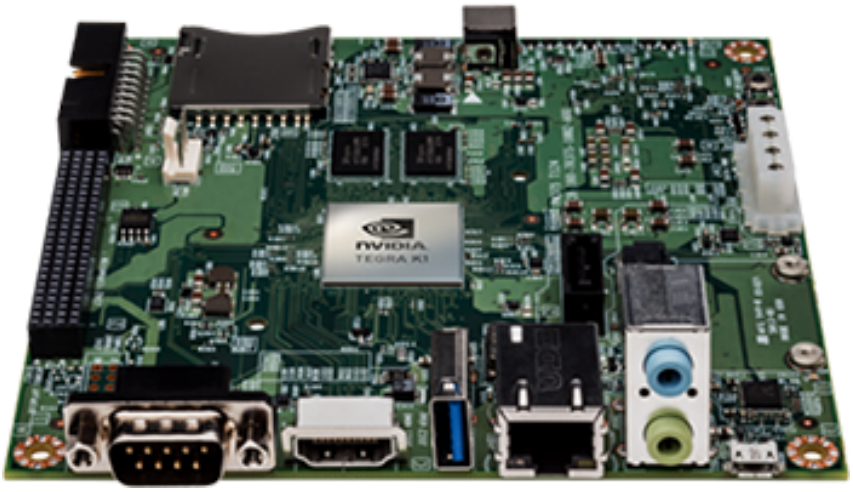


Figure 2.8: Picture of a Jetson TK1 development board [9].

platforms based on their volume of market; they can not compete against standard HPC products.

Chapter 3

Severo Ochoa Applications and Benchmarks

In this chapter we present the applications and benchmarks we used in our experiments to evaluate our prototypes and mini-clusters. First, we describe the in-house BSC production scientific applications included in the Severo Ochoa programme, and second, the set of benchmarks that compose the three layers of our platform comparison methodology.

3.1 Severo Ochoa Programme applications

The scientific applications included in the Severo Ochoa Programme are the following: Alya RED[6], from CASE department; NMMB-CTM[7], from Earth Sciences department; SMUFIN[8], from Life Sciences department; and Saiph, from Computer Sciences department. Table 3.1 shows a summary of the main characteristics of these applications.

Application	Model	Domain
Alya RED	Fortran&MPI	Biomedical mechanics
NMMB	Fortran&MPI	Weather Model
SMUFIN	C++&pthreads&MPI	Sequence Alignment
Saiph	C++&OmpSs&OpenCL	DSL for PDEs

Table 3.1: Severo Ochoa applications summary.

3.2 Alya RED

3.2.1 Application description

Alya RED is a biomedical mechanics simulator used for research in biological systems: cardiac models, cerebral aneurysms and human respiratory system among others. Alya RED is based on the parallel multiphysics code Alya, developed at BSC and it is part of PRACE Unified European Applications Benchmark Suite. Designed to run with high efficiency in large scale supercomputers, it has reported extremely good scaling results up to 100.000 cores in Blue Waters supercomputer, proving the viability of engineering simulation codes in exascale systems[23].

3.2.2 Implementation comments

Alya RED is written in Fortran and runs in parallel using MPI to split the workload along the nodes; uses a master/slave parallel model. To enable high parallelism and scalability, it uses non-blocking MPI calls, allowing overlapping of computation and comunications. Similar to a stencil code, Alya RED simulations are divided in iterative steps; every step advances a fixed amount of simulation time. Although in our experiments we disabled the output of binary files, Alya supports serial writing to disk using its own binary data format and parallel writing to disk using HDF5 data format. By default, Alya RED only depends on a third party library (metis-4.0), which is used for partitioning of the problem, provided with the package and

statically linked.

3.2.3 Inputs and experiments descriptions

All timings we show in the results section belong to the parallel iterative part, excluding initialization and finalization, running 10 timesteps. For energy measurements, the system reports energy values for the whole execution. To obtain meaningful energy measurements of the iterative part we run the simulations for 100 iterations, that way the iterative part represents around 90% of the execution time. Output to disk of intermediate and final results is disabled due to instability in the Mont-Blanc prototype. The data input we use in our experiments represents a rabbit heart beating.

3.3 Non-hydrostatic Multi-Scale Model on the B grid

3.3.1 Application description

NMMB is designed to be a flexible, state of the art atmospheric simulation system that is portable and efficient on available parallel computing platforms. NMMB is an evolution of the operational Non-hydrostatic Mesoscale Model (WRF-NMM), and it is currently used to obtain mid-range weather forecast predictions in the United States. In the last years, researchers at BSC, in conjunction with NOAA (National Oceanic and Atmospheric Administration) developed NMMB/BSC-Dust, an atmospheric dust model. Another NMMB version this time including a Chemical Transport Model [12] is also being developed at BSC. The code version we run in our experiments includes this last feature and it performs chemical transport forecast in addition to the regular weather forecast computation.

3.3.2 Implementation comments

Although new code is written in Fortran 90 and it uses MPI as a parallel programming model. NMMB partitions the earth surface in *tiles* and assigns each one of those tiles to a MPI Rank. Such tiles are determined manually configuring three parameters of the application: i, j and k; i especifies the number of horizontal tiles, j the horizontal and k the number of MPI ranks in charge of writing intermediate results to disk. That is, in a 64 MPI Rank weather forecast simulation we need to satisfy: $i * j + k = 64$

3.3.3 Inputs and experiments descriptions

In our experiments we perform 24 hour weather forecasts using a low resolution input set which allow us to perform basic simulations at a global scale. In other words, low accuracy 1-day weather forecasts of the whole world. Our timing results are obtained measuring the whole duration of the execution of the application. During the development of the project, we tested up to three different input sets, but as we explain in section 4.1.2 we could not run them due to memory allocation problems. Also in this case, during our test we disabled ouput to disk of results. We do not report energy measurements for this application.

3.4 SMUFIN

3.4.1 Application description

Somatic Mutation Finder (SMUFIN), is an implementation of the sequencing method with same name. Specifically, this method is able to achieve high throughput characterizing somatic structural variants while still obtaining high sensitivity and specificity values (which are the quality parameters in sequencing methods). Its main characteristic is that directly compares sequence reads from normal and tumor genomes, avoiding comparing sequences against the reference human genome. This application has been developed by the Computer Genomics group at the BSC Life

Sciences department and is also included in the Severo Ochoa programme. Although this application is not currently used in ṕroductionít is included in such programme based on its scientific relevance[20]. Computer Genomics group developed this application with two objectives: first, to validate SMUFIN as a bioinformatics method; and second, to open the door for a future tool to enhance personalized medicine diagnostics.

3.4.2 Implementation comments

The version of the code we analyze in this work was developed during 2014 and it is written in C++ using phtreads and MPI to parallelize the computation in tasks. Although the implementation generates correct results, we observed poor scaling performance while testing the application in MareNostrum3. In 4.2.4 we show a detailed analysis of this implementation and reason about the possible causes for this issue.

3.4.3 Inputs and experiments descriptions

The input used for our experiments is an *in-silico* genome sequence of the chromosome 22. In this case, the set of files required to sequence this genome takes up to 17 GB of space in disk. Timing measurements correspond to the whole duration of the application execution. At the end, the application generates a text based report indicating the mutations found in the sequences. In our tests we do not disable the output, it has a negligible impact on the application performance.

3.5 Saiph

3.5.1 Application description

Saiph is a domain-specific language (DSL) for solving Partial Differential Equations (PDEs) implemented as an extension of SCALA. Compiling codes written in Saiph language generates an intermediate code written in C++ plus OpenCL with OmpSs pragmas. Our performance analisis work starts after this last step, we compile those intermediate codes using the Mercurium [29] compiler obtaining executable binary files. Finally, we run these binaries to perform convection and diffusion simulations.

3.5.2 Implementation comments

As it is usual, the intermediate generated codes use variable names based on incremental numbers e.g: *x0, x1, x2...* and so on; those codes are extremely difficult to understand without knowing the original algorithm code.

3.5.3 Inputs and experiments descriptions

In our experiments we use three different Saiph generated kernels to simulate bitumen, convection and diffusion physics. Input parameters and input values are defined and generated inside the code, we use the default values provided there.

3.6 Benchmark layers

This section shows a brief description of the benchmarks included in the layers of our platform comparison methodology. Further, in section 4.1.1, we discuss about the objectives of each layer.

3.6.1 Layer 1

ARM FPU Benchmark It consists on two microkernels that we developed at the Mont-Blanc team embedding ARMv7/8 and ARM NEON assembly instructions into a C program. This kernels execute 50 million iterations of 32 scalar floating-point or SIMD instructions without dependencies between them, aiming to achieve the peak floating-point performance of the SoC. It reports average and individual GFLOP/s per core.

STREAM Proposed in 1995 by John D. McCalpin, STREAM is a benchmark for measuring sustainable memory bandwidth[35]. In order to stress the memory subsystem it uses datasets way bigger than the standard cache sizes. Executes four vector operation kernels: copy, scale, scale sum and triad using 8-byte size variables.

MultiPingPong Custom simple kernel developed at BSC, which tries to saturate a network link between two nodes by performing *PingPong* MPI communication in pairs where each processor in a pair belongs to a different node.

3.6.2 Layer 2

MILCmk, AMGmk and GFMCmk these codes are selected as a part of the benchmarking tools to evaluate machines in the CORAL project[14]. They are based on the most common algorithms used in full scale production applications that solve problems in areas that require to compute linear systems in unstructured grids or quantum chromodynamics simulations. They are implemented in C or Fortran and they run in parallel using OpenMP.

LULESH mini-app developed at LLNL (Lawrence Livermore National Lab) to perform simplified hydrodynamics stencil calculations, however LULESH represents the workload characteristics of more complex hydrodynamics codes. Uses both MPI and OpenMP to achieve parallelism.[16] [15] [13]

miniFE mini-app developed at Sandia National Laboratories included in the mantevo benchmark suite. Represents the computational behavior of the finite element generation, assembly and solution for an unstructured grid problem.

It is written in C++ and supports hybrid MPI plus OpenMP parallel execution.[18]

CoMD mini-app developed at LANL (Los Alamos National Lab) and also included in the mantevo benchmark suite. Used as a mockup of the typical molecular dynamics applications used in material science. It is written in C and runs in parallel using MPI.

HPCG High Performance Conjugate Gradient is a benchmark developed as a partial substitute of the High Performance Linpack benchmark. It aims to represent a wider range of computational behaviors that are usually found in current important scalable applications. Executes: Sparse matrix-vector multiplication, sparse triangular solve, vector updates, global dot products and a local symmetric Gauss-Seidel smoother. Written in C++ with MPI and OpenMP support [11] [4]

3.6.3 Layer 3

At this time, we only include Alya RED as a full production application suitable for comparing platforms (see section 3.2).

Chapter 4

Test and results

Following, we show the most relevant results we obtained during the testing and evaluation of the platforms. While doing so, we reason about the differences in performance, we discuss the issues we encountered and share our experiences and lessons learnt. Although the real dynamic of work was iterative, next sections are structured to follow an order based on the logic development stages of a prototype: port, test and optimize. In reality, we had to often revisit our experiments and apply modifications based on new information and findings obtained from other tests or other actors involved in the project.

This chapter is structured as follows: first, we share our experiences while porting the applications to ARM 32-bit and 64-bit architecture platforms; second, we present the performance results obtained with these applications in the Mont-Blanc prototype; third, we describe methodology to perform tests and performance comparison across mini-clusters; finally, we present the results and experiences using such methodology.

4.1 Experiences porting production applications

Running an application for the first time in a new architecture, specially in platforms in development, has a *high-crash* potential. Failure during compilation or execution

could be caused by missing dependences (e.g: our app requires BLAS libraries to run but they are not installed) and architecture dependent code among others. The actions we take around the application in order to execute and verify that it works correctly are part of the porting process of this application. Following, we share our experiences and describe our work done porting the Severo Ochoa applications to the Mont-Blanc prototype and mini-clusters.

4.1.1 Porting Methodology

Our methodology to port applications to a new platform is divided in three logic steps that we called: preparation, deployment and verification. During this process we keep track of all the actions required to work in the new platform and elaborate detailed installation guides. Those guides appear in Annex A

Preparation To prepare each application we first set up a startup meeting with members of the development team. The objective is to get to know first hand all the important information regarding the application in order to start working with it. Each team provides source code of the application (preferably a current working version installed on MareNostrum3), along with the installation steps, input sets and commands to execute it. Once we have this, we proceed to compile the code and execute it in MareNostrum3. This way we can check thath the application does not have any major issue on a x86 HPC system before porting it to ARMv7.

Deployment Next step is to deploy the application, once we have copied its files and before compiling, we install all the known dependencies (libraries and executables) based on the information obtained in the previous step. At the same time, we adapt the makefile and configure files to be coherent with our new target platform, we also might get rid of platform dependent code at this point. If compilation is succesful we continue to prepare files for execution, if not we identify the causes, fix them if possible. We consider this step finished

when we are able to run the application on the target platform without major issues.

Verification During verification we try to ensure two things, first and more important, the results obtained are correct and later, application still works when running with real input sets or resource configurations. To ensure the correctness of results, we compare the output files with the ones obtained on MareNostrum3. We can compare them at binary level using the linux command *diff*, or at visual level using visualization tools (e.g. Paraview, Ncview).

In our experience, all the applications we had to port are complex, highly configurable and different in usage. Therefore, all the information obtained from the developers team is key off to a good start. Also in our case, the follow up meetings were very productive for faster issue solving and better understanding of the application behavior. Although the step consisting of running and testing in MareNostrum3 is not sctrictly necessary, dealing with complete new production scientific software requires a process of learning its environment: inputs, outputs, configuration files, execution flags, visualization tools, etc. Application misbehavior can be caused by technical issues at many levels in an HPC cluster, better understanding of the application running in a stable platform helped us to identify faster at which level are the problems located.

During the porting process, we tightly collaborated with the Mont-Blanc sysadmin team to install the dependencies of each application. To avoid stall time in our work waiting for the dependences to be installed, we worked concurrently on porting several applications. Although this requires extra effort tracking the *in-flight* porting status of applications it allows the sysadmin team to establish a better work planning, resulting in an overall faster deployment.

4.1.2 Porting issues at ARMv7 Mont-Blanc prototype

In this section we describe in detail the specific technical issues encountered when porting applications to clusters based on ARMv7 architecture, i.e. the 32-bit instruction set of ARM.

Slow compilations We compile the codes and dependencies of each application in the Mont-Blanc prototype using the cluster login nodes. The first observation that can be reported is related to the average compilation time compared to MareNostrum3: on the Mont-Blanc prototype the compilation time are 4 to 10 times longer than on MareNostrum3. Although we do not report the experiments, we found that Lustre file system was the main cause of this bottleneck.

In order to reduce the long compilation times, we attempted to set up a cross-compile environment in a x86 server to build ARMv7 binaries. However a set of issues in deploying critical tools (i.e. gfortran and OpenMPI compilers) forced us to abandon this effort.

Alya RED Alya is the first application we started porting. Compiling and running this application is straightforward because it only has one self-contained dependency, besides the MPI runtime libraries.

We encountered two minor issues that required to be fixed during the port of this application. First, we found one instance of an intrinsic function *INT4()* which is only available in the *ifortran* compiler. We replaced this function with the standard intrisic form *INT()* available in both compilers. Second, we encountered a type mismatch in Alya's code when enabling HDF5 support. A function expecting an 8-byte integer was receiving a datatype that in 64-bit platforms is in fact 8-byte, but in 32-bit platforms is 4-byte long. This caused the compilation to fail in the prototype but not in MareNostrum3.

Sanity checks on this application are performed by visually comparing the results of the simulation in Paraview.

NMMB This is the largest and most challenging application in terms of complexity, number and size of files and thirdparty dependencies. In our experience, the gfortran compiler is more restrictive in terms of line length syntax than his

intel counterpart. As adapting length line of the whole code was not viable, we added the gfortran line-length control flag in almost every makefile. Also, the gfortran compiler flags that define the datatype lenghts during the preprocessing require special attention as they do not have an exact equivalent flag to ifortran.

In conjunction with the Earth Sciences NMMB-CTM team we developed a ARMv7 compatible version of the code, make and config files. As consequence of this effort, the fixed files were given to the NCEP developers and included in the next version release of the application.

When we tested the application with real input sets, we experienced a critical issue preventing us to run simulations with high resolution, like for example, world size simulations at 10km resolution. When using such inputs, the process in charge of managing the memory space of the output filegARMv7s was trying to allocate a number of bytes in memory that was exceeding the biggest integer number representable on 32 bit integer. This was causing an integer overflow exception and making the execution to fail.

This problem is of course not visible on 64 bit architectures because integers can be represented using 64 bit and this variable size is enough for mapping the memory size needed by the application.

Theoretically, this issue could be addressed parallelizing the memory allocation for I/O, however NMMB relies on only one MPI process for I/O requiring major changes in the code for the implementation of this solution. For this reason, we had to leave larger cases not-tested on 32 bit platforms.

As more general comment, we can expect several other scientific applications requiring the allocation of arrays with a number of elements larger than $2^{32}-1$. This also implies the need of a larger address space addressable by one process. And finally this is one of the main reason because scientists tend to assume that real HPC workload is performed using 64 bit architectures. We know

however that in some case this limitaitons could be avoided with a better design of the applications.

To validate the correct execution, we transform the output files generated during execution to NetCDF data format and visualize them with it is default viewer, NCView. Figure 4.1 is an example of our view when comparing MareNostrum3 and Mont-Blanc prototype ouputs.

(a) MareNostrum3 (b) Mont-Blanc prototype

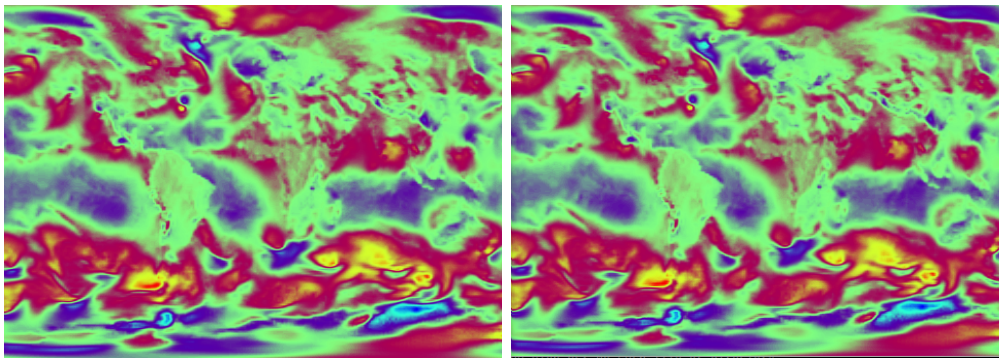


Figure 4.1: Image of NMMB output files opened in NCView.

Saiph During the porting phase of this application we reported a bug in the Mercurium compiler when dealing with *C++ Templates*[30]. We also experienced issues with the OpenCL libraries that caused wrong results when performing floating point operations in the Mali GPU. Both issues were fixed in collaboration with the Programming Models team at BSC and the Mont-Blanc sysadmin team respectively. Finally, we adapted the work_group_size parameter, the Mali-T604 GPU available on the Mont-Blanc prototype do not support in fact values higher than 256.

In this case, to validate the results we used MinoTauro Cluster, a x86 platform with NVIDIA CUDA GPUs.

SMUFIN We received the code of the application knowing that the code was depending from the BWA library that used x86 SIMD Intrinsics. Here, we de-

scribe which approaches we took to fix or workaround this problem, and why they did not work.

First, we analyzed the previous and newer versions of the BWA library. Previous versions allowed to disable SIMD intrinsics at compile time, but downgrading to older versions was not possible since those versions did not implement some of the functions the functions needed by SMUFIN. Newer versions were also x86 dependent. Second, we searched for alternative implementations of the BWA functions without success. We finally had to discard the idea of a quick partial porting of the key functions to the NEON SIMD and ARMv7 FP instructions due to our lack of knowledge in sequence aligment methods. Because of that, we were not able to succesfully port this application on ARM based platforms. Instead of performance tests and results in Mont-Blanc, we report an analysis of the parallel issues of this application based on the source code and the traces obtained in MareNostrum3 (see section 4.2.4).

4.1.3 Porting issues at ARMv8 platforms

In this section we report our experiences porting scientific applications to ARMv8 based platforms.

As a general comment, the deployment of both applications and the software stack was way faster than the previous one in Mont-Blanc prototype. In our opinion, the main reasons causing such *speedup* are the following: first, the experience adquired during the Mont-Blanc prototype deployment; second, the improved support for ARM architectures in the HPC ecosystem; and finally, the fact that ARMv8 is a 64-bit ISA, that avoids several of the conflicts and datatype mismatchs encountered in 32-bit architectures while installing and running applications and libraries.

From the four Severo Ochoa applications we decided to port and test Alya RED and NMMB, the two most promising and active applications.

As we have seen in the previous section, Alya RED did not require major port-

ing actions to run properly in Mont-Blanc prototype, and in our experience, neither is the case for ARMv8. NMMB on its side required two minor actions. First, a dependence had its *configure* files generated using an autoconf version without support form ARMv8; official forums offer technical support for this issue[33]. Second, we had to add support for ARMv8 architectures for some parameters in NMMB is framework libraries that were under DEFINE clauses which did not contemplate the ARMv8 option.

We did not experience other impediments while trying to install and run other benchmarks and mini-apps. In our opinion, currently the HPC software stack ecosystem support for ARMv8 is highly satisfactory.

Further, in section 4.3.1, we test the ARMv8 platforms using different applications than on the Mont-Blanc prototype.

4.1.4 Compiler comparison: intel vs gcc

For intel x86 based HPC clusters, Intel provides its own compilers, MPI and linear algebra libraries. Such implementations are not open source, and obviously neither they support ARM architectures. As Intel HPC software and compilers are known because its good performance, we attempted to measure the potential performance losses caused by the use of software alternatives. In this section we compare the performance compiling NMMB using GCC (i.e. gfortran + OpenMPI) versus Intel (i.e. ifortran + OpenMPI) in MareNostrum3.

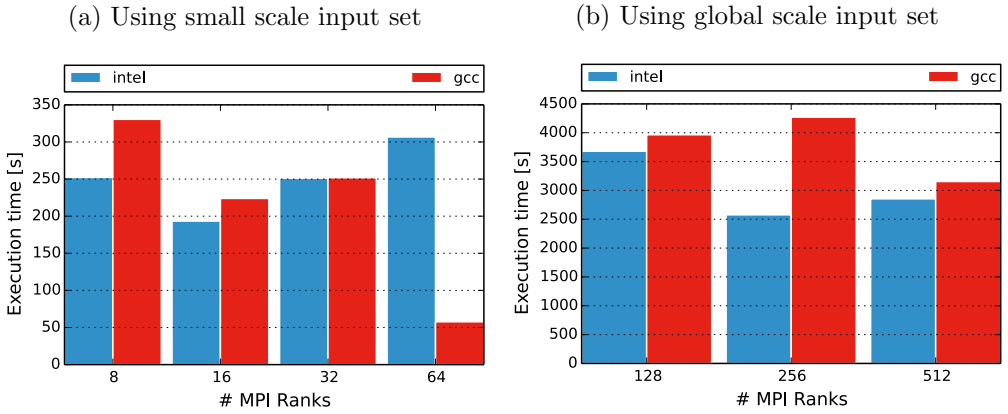


Figure 4.2: Performance comparison executing NMMB built with Intel or GCC

We build the same sources using two diferent compilers but linking the same MPI runtime libraries (OpenMPI v1.8.1). The Intel version of NMMB is built using *ifort 13.0.1* and the *GCC* version is built using *gfortran 4.9.1*. In figure 4.2 we present the performance results executing NMMB using small scale (4.2a) and a big scale (4.2b) input sets on both versions. In (a) we observe an irregular behavior showing from 30% speedup using *gcc* up to 6x faster execution time using *ifortran* with 64 cores. Global scale input set shows performance results in favor of *ifortran* with speedups ranging from 10% up to 60%.

As we said, big production applications not always exhibit good parallel behavior, being NMMB one of those applications. Because of that, although this comparison shows results in favor of Intel compilers, the fact that we are such unstable application makes them far from conclusive. Based in this experience with NMMB, future expansion of this study would require the use of a wider set of benchmarks as a tool to quantify the performance impacts. For example, we should find at least one production application that shows reasonable parallel scaling behavior, like Alya RED; and a set of miniapps which are more flexible but still representative of workload characteristic of full production applications. Besides the compiler comparison, we should also expand the study to consider linear algebra libraries, like MKL versus ATLAS; and MPI implementations, like Intel MPI versus OpenMPI.

4.2 Mont-Blanc performance results

We present in this section the computational performance and energy efficiency results obtained on the Mont-Blanc prototype. As we mention before, the Mont-Blanc prototype experienced many changes along the months of development of the project (at hardware, system software and application level). This changes cause variations in the performance on the cluster, therefore giving relevance to the timing of each experiment. In our results we reason about how the current state of the cluster affects our experiments.

4.2.1 Alya RED

In this section we show results obtained evaluating the prototype using Alya RED. We present the best results up to date in the final results section. We also show some intermediate results obtained in previous months to be able to discuss about the progression of stability and performance during the development of the prototype.

Final results

In this section we show the final performance (4.3, 4.4a) and energy (4.4b) results obtained on the prototype (MB) and MareNostrum3 (MN3) in late February 2016. As input set, we use a rabbit heart model with a resolution of 2.6 Million elements and run the simulation for 100 time steps.

Figures 4.3a and 4.3b shows that we can scale the application up to 512 cores with 99% efficiency and up to 1500 cores with 70% parallel efficiency.

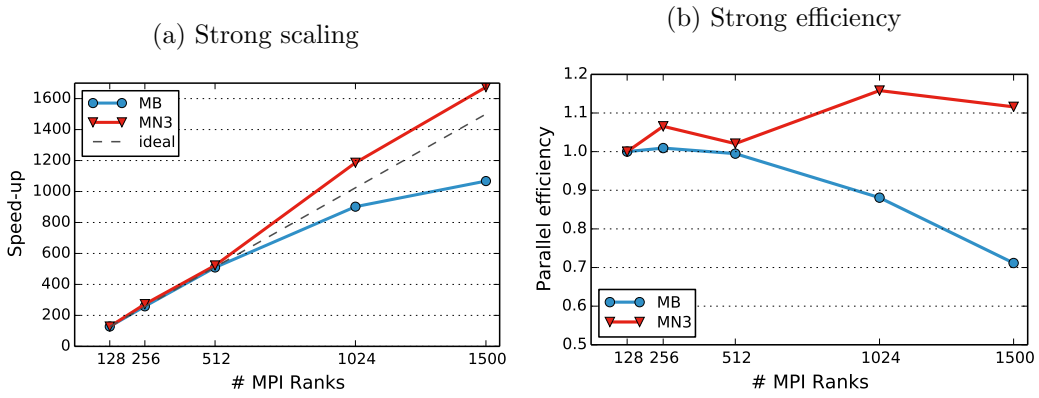


Figure 4.3: Mont-Blanc and MareNostrum3 parallel speedup and efficiency running Alya RED

The parallel efficiency over 100% on MareNostrum3 when increasing the number of cores suggests that, in Alya, we obtain locality benefits when each processor has to compute smaller parts of the global solution.

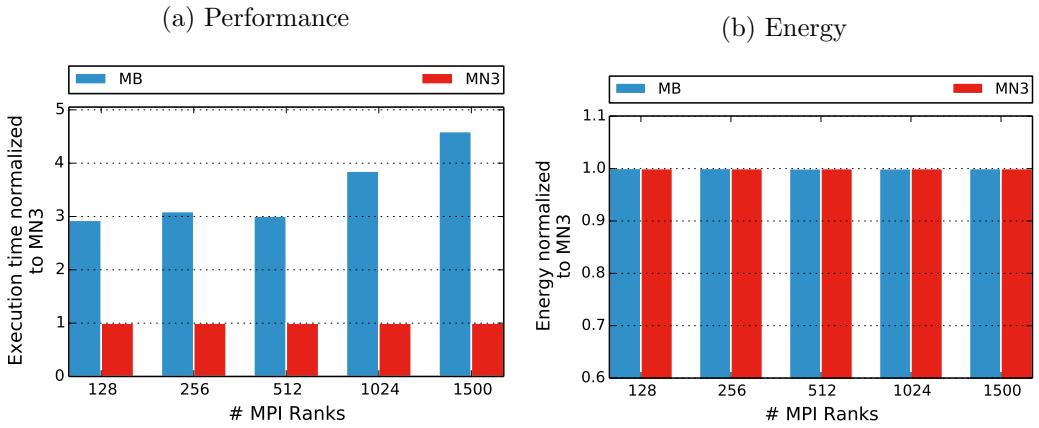


Figure 4.4: Performance and energy comparison

Figure 4.4a shows the performance of Alya on Mont-Blanc prototype compared to MareNostrum3 supercomputer. When using the same amount of cores, Mont-Blanc is 3x to 4.5x slower.

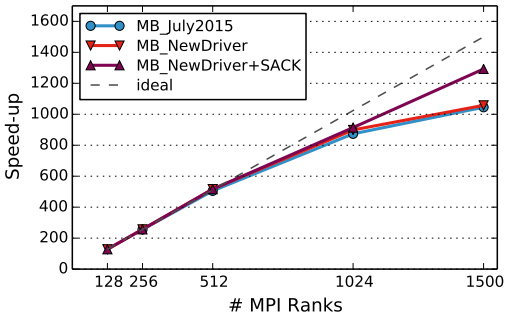
In figure 4.4b we compare the total energy consumption (Joules) per execution. The results show that in this case, energy consumption is very similar when using the same amount of cores in both platforms. In order to obtain relevant energy measurements, each execution runs for at least 10 minutes so we minimize the impact of the initialization and finalization phases.

Evolution of performance

In this section we depict the evolution of results obtained on the Mont-Blanc prototype along the months of development.

After July 2015 the prototype underwent several upgrades in the network stack. In early november 2015 we repeat some experiments to test the cluster performance after the network improvements. During these tests we encountered additional critical issues causing performance loss that were related to the network configuration. Once the problems were identified and fixed, we repeat the same experiments. In section 5.1.1 we describe in detail these issues and the actions required to fix them.

(a) Strong scaling



(b) Strong efficiency

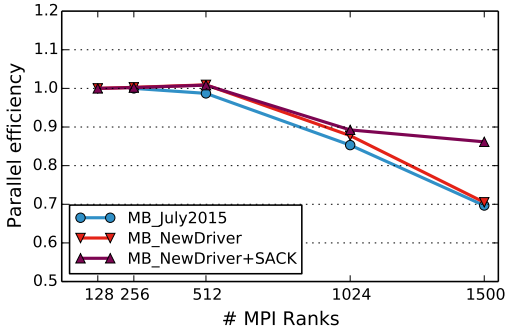


Figure 4.5: Performance and energy comparison

Previous figure (4.5) shows the scaling results after each network stack upgrade of

the prototype. The line labeled as *MB_July2015* represents the results gathered during Summer 2015, *MB_NewDriver* represents results obtained in november 2015, *MB_NewDriver+ SACK* represents the results presented during the review of the project in december 2015. In this executions we use the same input set and we run for 10 simulation timesteps.

As we see in figure 4.5a, compared to July 2015, in November we do not obtain any significant scaling benefits from the network upgrades. Although those upgrades improved the perfomance of several other applications when using high number of cores (> 512), Alya parallel efficiency was virtually unaffected. The plot in figure 4.5a is an example of the many technical setbacks we experienced during the development of the project.

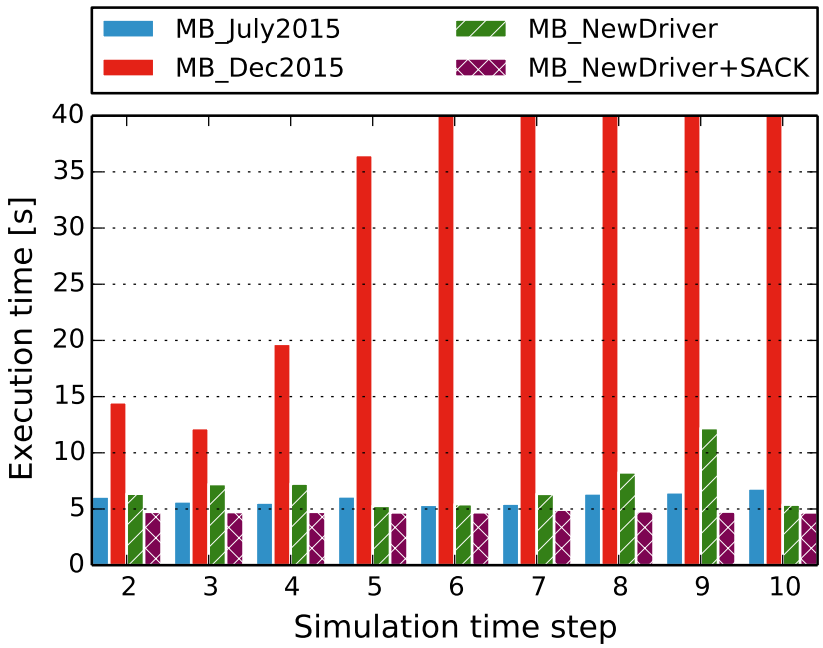


Figure 4.6: Alya RED perfomance evolution

In our experiments, all the timesteps of a simulation should take the same amount of time (variability under 10%). We now analize the stability of the cluster performance after applying each network improvement. In figure 4.6, we show the individual duration of each timestep inside a simulation. As we mentioned, in november prior to

the review of othe project, we experienced important performance issues; bars cut in the plot represent iterations of 121 seconds duration. We observe timesteps (see *MB_Nov2015* results) that range from 3x to 25x slower than in the previous experiments in July. Again, we see that after the first upgrade we improved the results and reduced heavily the timestep duration variability of simulations, although the results were still worse than in July. Second upgrade (*MB_NewDriverSACK*) not only fixed completely the variability issues in Alya, it also caused a 1.25x speedup in the timesteps.

Note that in terms of speedup we seemed to obtain better results at the end of December 2015 (4.5a purple line) than in February 2016. This is due to a change introduced on the timing function of Alya before the February experiments. Since the beginning of the experiments up to the end of 2015, we used a measuring function based on the elapsed *cputime*, meaning that we are not taking into acount System time in our measurements. Opposed to MareNostrum 3 nodes, Mont-Blanc nodes do not include a DMA support, therefore, the system is not capable of offloading network transfers. A DMA engine allow us to send messages over the network without active usage of the CPU to copy the buffer from memory. The original Alya RED timing functions are based in CPU time instead of Wall Clock time, that is a potential source of unfairness when measuring iteration time, because of that, we modify the timing functions to measure the time based on the Wall Clock time. Using this new version, we obtain worst scaling at high number of processors, where potentially the overhead of the operating system is more significant.

4.2.2 NMMB

As we present in the porting experiences (see 4.1.2), due to critical issues we are not able to execute global scale simulations in the Mont-Blanc prototype. All the results we show in this section are obtained using the low resolution inputset (basetest). Even if this input is not relevant for scientific experiments, it still remains representativo for studying the behaviour of the application at scale.

Final results

Following we present the final results of NMMB. The power monitor was disabled at the time of these experiments, thus, we do not show any power consumption results for this application in the Mont-Blanc prototype.

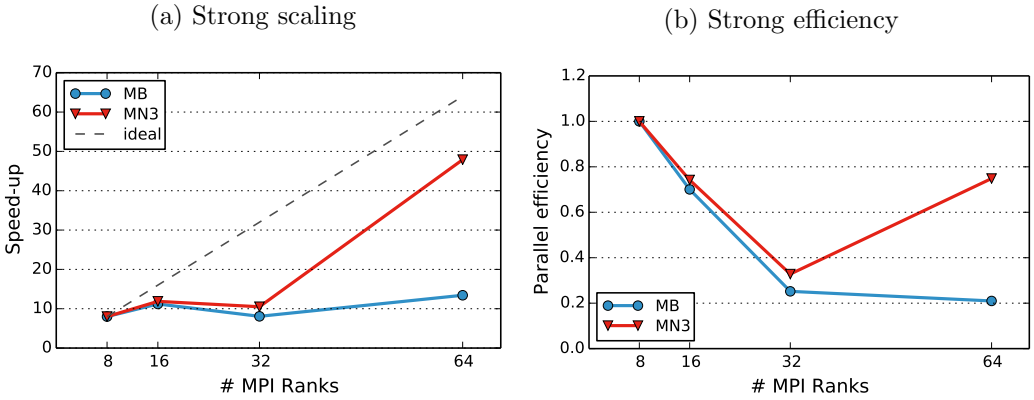


Figure 4.7: NMMB parallel performance

In Figure 4.7, we present strong scaling performance results running NMMB using up to 64 cores to perform weather forecast simulations. We already observed the bad scalability of NMMB in a previous section (see 4.1.4). Also in this case, the parallel efficiency drops quickly below 40% showing that there is almost no improvement in performance when we increase the number of MPI Ranks in both platforms. At 64 MPI ranks MareNostrum3 improves heavily its performance achieving up to 80% parallel efficiency. We are not able to identify the causes of this performance improvement, to do so, as a future work we should study the differences in performance based on traces by observing the evolution of the different computational phases on strong scaling tests.

In Figure 4.8 we compare again the execution time normalized to MareNostrum3. In NMMB the Mont-Blanc prototype is 5x to 19x slower, being 9x slower in average.

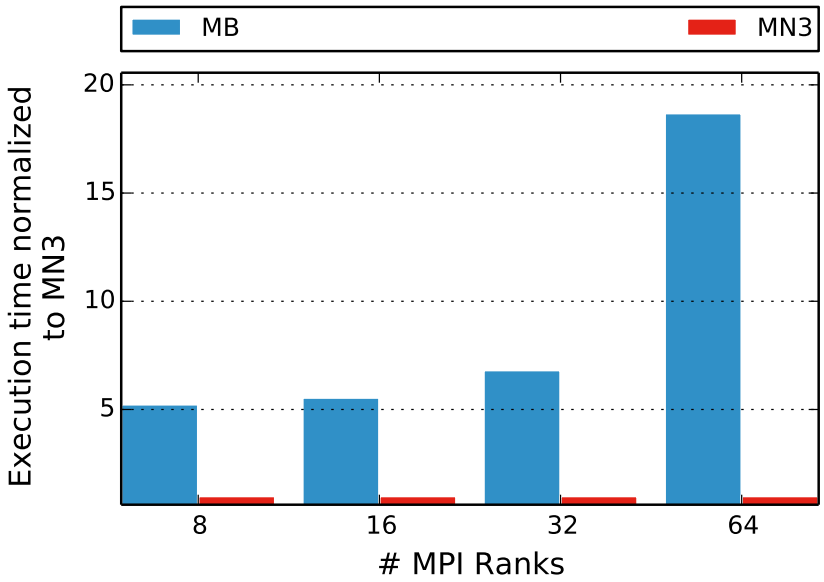


Figure 4.8: NMMB performance on Mont-Blanc compared to MareNostrum3

In our experience, NMMB can be hardly used to compare platforms in terms of scalability. Dedicating high quantity of effort to run NMMB in the prototype barely produced any results in terms of performance numbers or figures.

4.2.3 Saiph

Following, we present the results we obtained testing the prototype with CFD codes generated using Saiph. As we said, Saiph generates OpenCL kernels capable or running on GPUs. The execution time of these codes is spent in the OpenCL kernels; in the GPU. As its normal, our test scaling the number of CPU cores show no improvements in the execution time. Instead, we show a platform comparison enabling or disabling the output of the application to disk.

Final results

We test three kernels: Bitumen, Convection and Diffusion2Eqs. The input set parameters are *embedded* in the source code and we do not perform any tuning on them. We compare the results against MinoTauro, the BSC platform housing 61

nodes with dual socket Intel 6-core processors and 2x Nvidia Tesla M2090 accelerators.

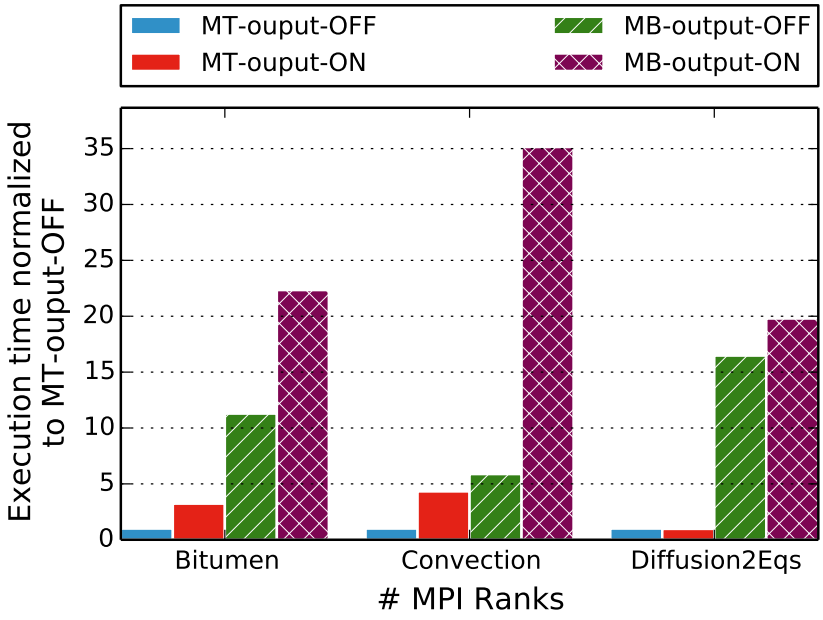


Figure 4.9: Saiph performance on Mont-Blanc compared to MinoTauro

Looking at figure we observe two things: the GPU performance gap between platforms, and how it relates to the disk writting of the output. In terms of GPU performance, the Mali-T604 is 6x to 16x times slower. In MinoTauro, compared to disabling the output to disk, enabling it causes slowdowns up to 4x. In the same circumstances Mont-Blanc prototype is 1.25x to 6x times slower.

In our experience, writing files to disk in the Mont-Blanc prototype is more expensive than in other conventional HPC cluster. In this results, the performance gap between GPUs is relatively bigger than the Cortex-A15 vs Intel Sandybridge gap. As a final comment, since we are focused in the scaling of scientific production applications, we just performed a preliminary study on the Saiph kernels, as they are not suitable for scalability tests on the full cluster. However, we consider that an OmpSs ad Cluster (an OmpSs version with distributed memory support) version of this codes would be an interesting benchmarking tool because it would

allow to easily evaluate new physics codes using the full cluster with optimized code.

4.2.4 SMUFIN

As we mentioned before (see 4.1.1), it was not possible to port SMUFIN to the Mont-Blanc prototype. Instead of the standard performance analysis and comparing Mont-Blanc prototype against MareNostrum3, we present an in-depth analysis of the scaling behavior of the application. To do so, we analyze the scaling and memory consumption of the application; and also, we share insight about the behavior of the application using paraver traces. At the end of this section, we discuss the overall problems of SMUFIN and add some comments related to the impact of the collaborative work done around this application.

Scaling and memory consumption analysis

We run all our tests using the synthetic input set 'ch22_in_silico' which aims to represent the genome sequences of the chromosome 22. This input set is distributed at the SMUFIN website for evaluation purposes.

In our strong scaling tests we do not obtain almost any speedup. Also, reducing the input set length in about 80% causes only a 15% execution time reduction. This suggests two things. First, for the new method that SMuFiN describes (see 3.4), the BWA library for sequence alignment is not efficient because this library forces to load the human reference genome data ($> 10GB$) which is not used later. Second, the parallel implementation of this method has critical parallel efficiency problems.

Like several other sequencing applications, also SMUFIN deals with complex and large data structures translating into a large memory footprint. A regular execution aligning one chromosome (ch22) uses on average 46GB of RAM. In figure 4.10 we see that the total amount of RAM required to execute the same problem increases linearly with the number of processors, there is almost a 100% replication of data

structures between processes.

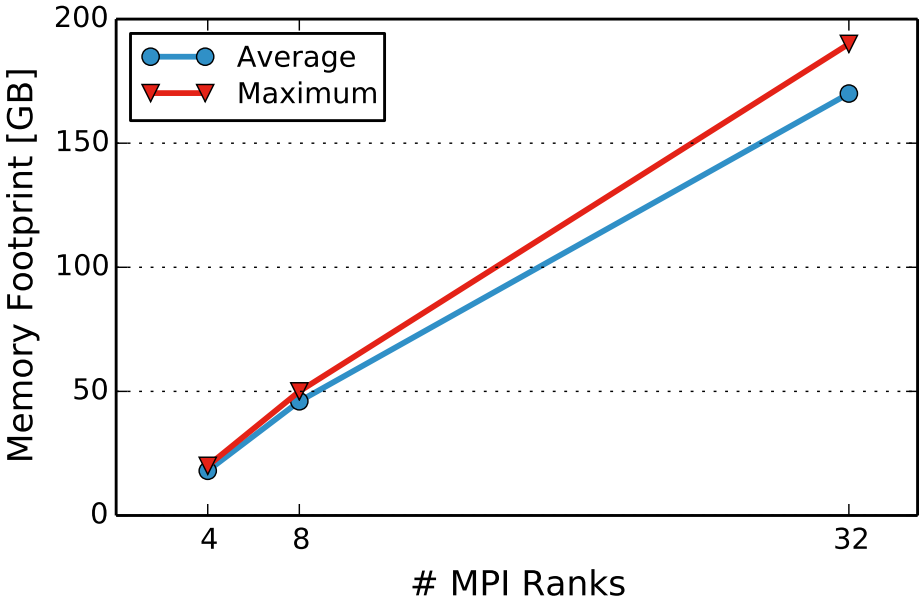


Figure 4.10: Memory footprint scaling the number of MPI processes.

Trace analysis

We use tracing tools (Extrae and Paraver) to visualize and analiyze the behavior of the application. The images we show here correspond to a trace obtained from a execution using 64 cores equally distributed between 8 nodes. Every figure correspond to a different view that renders different information. in further specific observations, we refer to the number annotations inside each image.

Figure 4.11 shows the useful duration of the computational bursts. Light/Green values represent lower values, dark/blue for higher values. In the figure we see two kinds of lines, horizontal (1) and stepped (2). After the first (1), each horizontal line represent a slave MPI process (1a, 1b) that manages a node.

In SMuFiN, when a node has an idle core, the slave process of that node request and fetch work blocks from the master MPI process. Then, the slave process creates

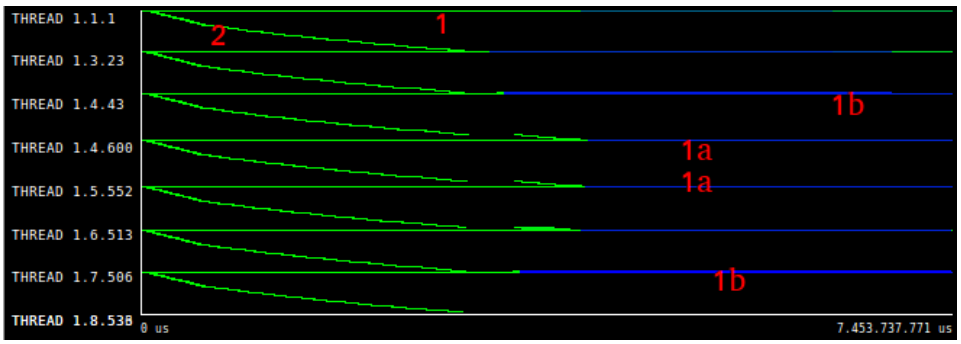


Figure 4.11: SMuFiN timeline showing useful duration of threads.

a thread to compute the block and gives it to the idle CPU. Upon finishing, the results are sent to the master and the CPU is again free/idle. In the stepped lines (2), each *step* represents a pthread performing the computation required for a work package for some amount of time (on that depends the lenght of the line) before being destroyed. As we see, everytime we create a thread its running only for a few milliseconds.

Below in figure 4.12, we see now a trace where each thread is colored by its *computational state* (legend right). We observe how the communication rate between master and slave is very intensive during the first half of the trace. Those comunications correspond to the slave processes fetching and returning work blocks to the master. After that, in the second half, there is only three events where slave processes send work to the master (circled in red). While waiting for this three work blocks to finish, all the other nodes finished its work and their cores are idle. This *outlier* work blocks take $\approx 200x$ more time to complete than the average.

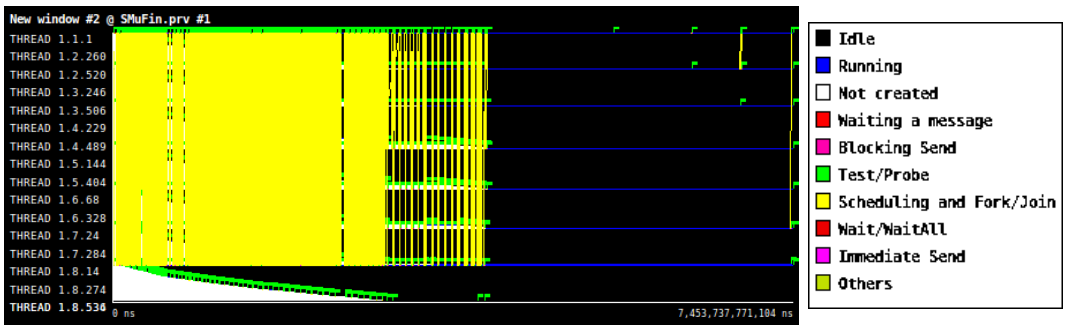


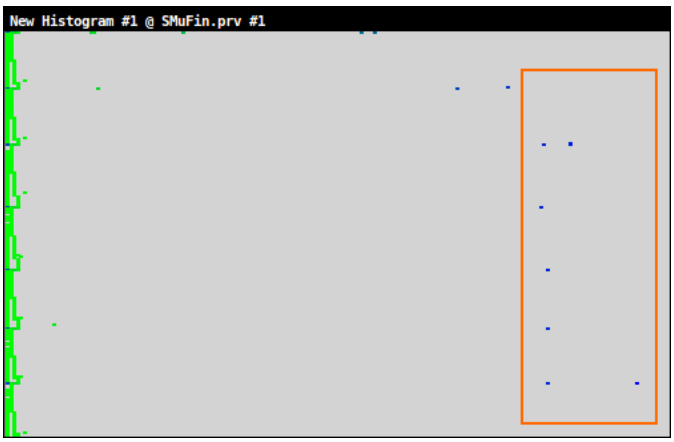
Figure 4.12: SMuFiN timeline view with communication lines.

In figure 4.13a, we show a histogram of the computational bursts duration. The region marked outlines a set of bursts that are exceptionally long. With paraver we *zoom* into that region, this way we can isolate the abnormal behavior bursts. Figure 4.13b shows only these marked objects in a view which has the same scale as the previous view in figure 4.12.

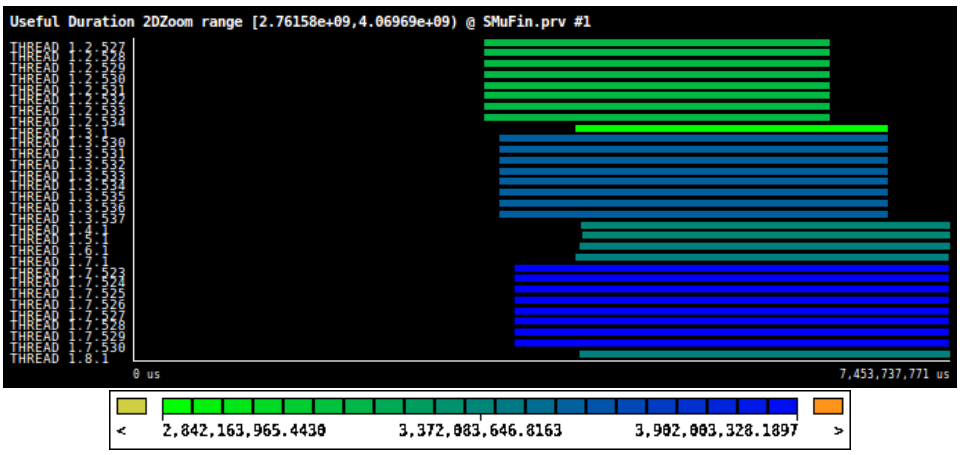
We observe two different groups. First, the group of slave threads (Thread 1.2.1, 1.3.1, etc), they are expected to be this long, as they are running during all execution till the end. The second group is divided in 3 subgroups of eight threads, each subgroup corresponds at the last work block computed at different nodes. If we extract the total instructions hardware counter values for these bursts we obtain that these bursts execute up to $\approx$ 180x more instructions than the average. At this point, finding the issue causing this behavior was out of the scope of our target work, since it required an in-depth debugging of the application and major changes in the code.

Overall issues and conclusions

Now we comment on the overall issues we found in terms of parallel performance and conclusions about this application.



(a) Useful duration histogram.



(b) Zoom of the outlier bursts we found in the histogram.

Figure 4.13: Analysis of the outlier work blocks

Parallel efficiency of this SMuFiN version is very poor, increasing the number of nodes beyond 8 or 16 nodes does not yield any benefit in performance; it does not scale. Trace analysis shows heavy load unbalance between nodes. On the other hand, we consider that a solution to the current distribution of the data structures in memory is required; it should not increase linearly with the number of processors.

We suggest reconsidering the parallelization approach of the application in favor of a more state of the art programming model with tasks runtime support in addition to MPI, that is, OpenMP 4.0 or OmpSs. Also, in the current implementation threads are managed by SMuFiN's own runtime functions that

are: first, inefficient to work with a complex workload like this one; and also, very difficult to analyze using tracing tools.

Inapropiate libraries Our test shows that changes in the input sets have low impact in the execution time. We are strongly convinced that this is caused by the BWA library used for sequence alignment. As we said, this library loads the full reference genome of 17GB when the original design of the method does not require reference genome comparisons at all. This justifies the effort to migrate to another library or create our own set of sequence aligment and management functions. Also, replacing BWA will remove the x86 intrinsic code, opening the possibility of a full port of the application to ARMv7/8

Interdepartamental collaboration The testing and evaluation of SMuFiN in MareNostrum3 and the report of results to the developers influenced several actions on the application development itself. As of May 2016, this application its currently being reimplemented from scratch, based on the collaborative work with several groups: Heterogeneous Architectures, Storage and Big Data. In this new application the developers are considering new approaches to tackle the problems identified by the collaborators.

In the future, Life Science group expects to obtain a production level scientific application based on SMuFiN capable of handling several full genome sequences at the same time.

4.3 Performance evaluation of ARM 64-bit platforms

As we mentioned, during the progress of the Mont-blanc project, the research and development team deployed several mini-clusters and one large HPC prototype (see chapter 2). Although the new platforms share majority of its software stack, they are very diverse in hardware. Each SoC has different configurations: number of cores and nodes, ISA (ARMv7-a/8-a), topology and network interconnection.

In the project, we wanted to perform a fair perfomance comparison of all our avail-

able platforms. For that, we design and develop a methodology to evaluate performance based on obtaining timings and metrics that allows us to compare the clusters and also specific parts of each platform architecture. We test and improve our methodology by using it to compare the last Mont-Blanc generation platforms: Applied Micro XGene, Cavium ThunderX and Nvidia Jetson TK1.

Below, we present and reason about our methodology and results evaluating these platforms. We also include an issues subsection to comment on the technical and performance problems we encounter during our tests. The porting experiences related to ARMv8 platforms are shown in the previous section 4.1.3.

4.3.1 Methodology

In order to perform a fair and complete comparison of performance running scientific applications in low power architectures, we cannot rely only in a single application. As we have seen in the previous section, full production applications generally are not designed to work as a benchmark. In consequence, adapting the parameters and inputs to perform a wide range of tests is very difficult. We realized that testing for HPC performance we need to use applications as close as possible to production scientific applications, but we need also to understand the performance at a smaller, detailed scale. Because of that, we decided to select a new set of benchmarks that will allow us not only to perform a fair comparison, but also, compare individually several points in the architecture of our mini-clusters. Finding applications that can run properly on every platfrom is not a trivial task because we are constrained by their heterogeneity. In the list of hardware characteristics of our platforms we find variety in the number of nodes, cores per node, main memory sizes, etc.

To select and organize the set of benchmarks that target different parts of the architecture, we decided to structure our benchmarks in layers. The objective of each layer is to gather a different set of metrics that allows us to fairly compare the performance of our platforms executing HPC applications. We develop or select the benchmarks that we consider more suitable for each metric. To fill each layer, we explore several state of the art scientific community accepted benchmarks like: NAS

parallel benchmarks [22], Polybench/C[28], Mantevo[18], Rodinia[36], CORAL[14], PARSEC[24], etc.

In the following sections we define three layers of benchmarking, including the objectives and the target study for each layer.

Layer 1 aims to individually test the architecture limits of a single node. More specifically, we execute benchmarks to study throughput and latency of three main component of HPC systems: the floating point unit, the memory subsystem and the network infrastructure.

Our approach in this layer is pretty straightforward, we use one single benchmark for each metric. First, to measure the maximum achievable floating-point performance we execute FPU benchmark.

Second, to test the memory bandwidth capabilities of our memory hierarchy we use STREAM. STREAM is a well-known and widely accepted benchmark that measures sustainable memory bandwidth (MB/s) executing four different vector operation kernels. Memory bandwidth measurements are key to study and understand the behavior of applications in multicore systems.

Finally, we test the maximum attainable network bandwidth (in Mb/s) running another custom benchmark, MultiPingPong. MultiPingPong tries to saturate a network link between two nodes by performing PingPong communication in pairs where each processor in a pair belongs to a different node. This also allow us to reason about the limits and behavior of the network performing demanding message passing operations.

Layer 2 aims to test the parallel perfomance of our platforms using scientific community accepted benchmarks. Among the set of benchmarks we have chosen, we distinguish three types: first, the miniapps CoMD, LULESH and miniFE; second, the standard HPC Benchmark suites HPCC and HPCG; and finally, the OpenMP microkernels MILCmk, AMGmk, GFMmk. With the first two

groups, we are able to test the performance of the full machine with applications designed to test parallel efficiency up to a high number of nodes. With the last one, we focus on the multicore inside a node with OpenMP kernels. That allows us to study the impact (coherence network, memory bus, NUMA penalty, etc.) and sources of overheads when running shared memory applications.

All the benchmarks we selected belong to state of the art benchmark suites. LULESH miniFE, MILCmk, AMGmk and GFMmk are part of the CORAL project benchmarks. CoMD is included in Mantevo benchmark suite. Finally, HPCC (which includes HPL) and HPCG are considered as they are used for international ranking of HPC systems.

Layer 3 aims to evaluate the full cluster scaling with state of the art scientific production applications. Applications in this layer allow us to stress all the HPC cluster architecture at once. Benchmarks in layers 1 and 2 are not designed to write to disk big output files while production application usually provide support for it, thus, allowing us to test also our storage subsystem behavior. Considering the complexity of porting and testing large production applications, we limited our study in this layer to Alya RED.

4.3.2 Layer 1 benchmarks results

Below, we present the Layer 1 benchmark results. From this tests, we are able to obtain the maximum values per platform of this following metrics: floating-point performance (FLOP/s), memory bandwidth (MB/s) and network bandwidth (Gbps). These values allow us to directly compare the capabilities of each socket; this is key to reason later on about the impact of running parallel benchmarks using all the cores in a socket or shared memory domain.

Floating-point performance comparison

To obtain the floating-point performance of each socket we run the FPU benchmark (see 3.6.1). Although we obtain single and double precision performance for both ARM VFPU and NEON SIMD units, in this study we only report double precision performance of the VFPU.

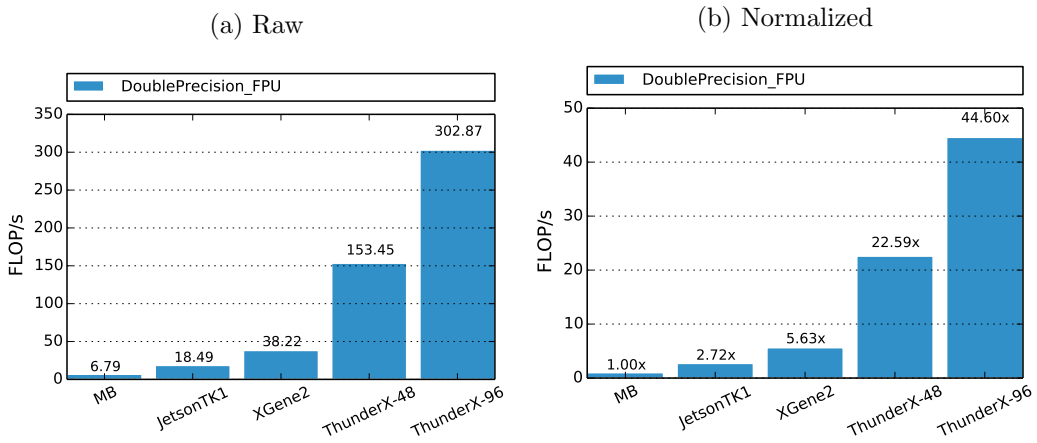


Figure 4.14: Double Precision Floating-point performance SoC comparison

Figure 4.14 shows, for each platform, the aggregated double precision floating-point performance of all cores in one node. For future references, we also show the normalized FLOP performance (4.14b). In the X axis of these plots, ThunderX_48 refers to executions using 1 socket (48 cores) of the two available in each ThunderX node, similarly, ThunderX_96 means we use both sockets. We use as baseline 1 Mont-Blanc prototype socket (MB) with 2 cores.

First, a JetsonTK1 socket has performance improvement of 2.72x over Mont-Blanc. Although both SoC's use the same Cortex-A15 implementation, JetsonTK1 has 4 cores instead of 2; running at a 2.32GHz frequency instead of 1.7 (1.36x improvement). Mutiplying Mont-Blanc performance value by both factors, number of cores and frequency increase, we obtain the same value that we obtained in the benchmark, 2.72x; which is in fact, expected. If we apply the same reasoning comparing JetsonTK1 and XGene2 we also obtain that the improvements in performance match

exactly the relative increment of number of cores and frequency. As we mentioned before (see chapter 2) both architectures have a throughput of 2 FLOP per cycle. However, this trend does not continue with the ThunderX socket, considering its frequency and number of cores and floating-point throughput we expected a 25.3x increase in performance.

Memory bandwidth comparison

In this section we present the STREAM benchmark results in the three platforms. Executions run an out-of-the-box version of STREAM and use all cores available in each socket (running with OpenMP).

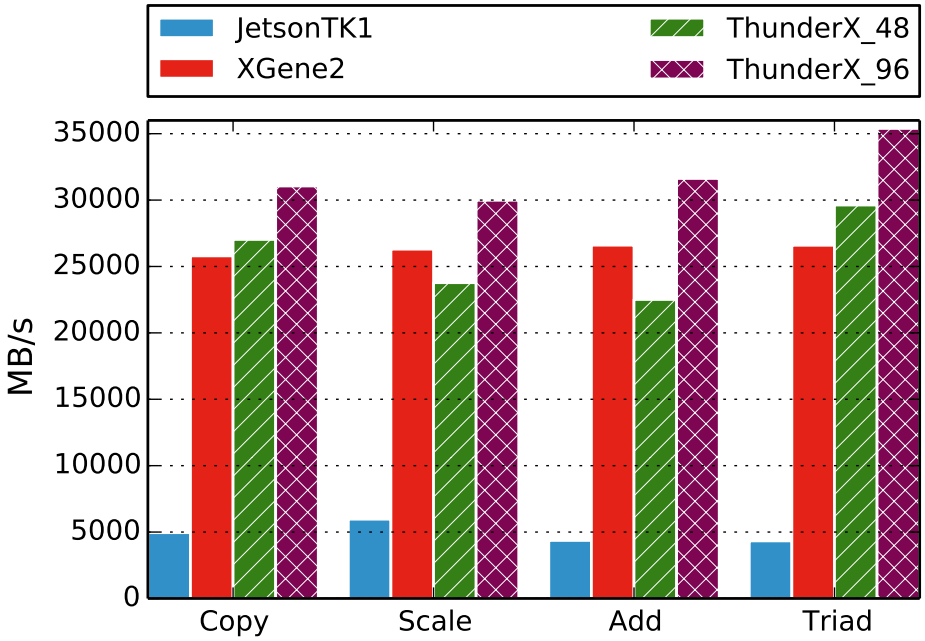


Figure 4.15: STREAM benchmark memory bandwidth results.

Figure 4.15 shows the maximum memory bandwidth achievable by each socket when running the four vector kernels of STREAM. Again, ThunderX_48 and ThunderX_96 refer to the use of one or two sockets on a ThunderX node.

Compared to JetsonTK1, XGene2 achieves 5x more memory bandwidth. ThunderX_48 achieves at least 20 to 25 GB/s of memory bandwidth; using both sockets in ThunderX_96 achieves 30 and 35GB/s. From this results we remark three things. First, with only 8 cores, XGene2 has approximately the same bandwidth compared to one ThunderX socket, therefore, XGene2 has 8 times more memory bandwidth per core when threads occupy all the cores in both platforms; this will favor XGene2 scalabiliy in memory intensive applications. Second, JetsonTK1 is far behind the rest of the platforms; in the previous section, JetsonTK1 compared to XGene2 achieved reasonable floating point performance results. The ratio memory bandwidth to computation decreases significantly meaning that even running applications with the same number of cores, JetsonTK1 would suffer earlier from memory bottlenecks. Third, when going up from 48 cores to 96 cores we experience a big drop in efficiency, doubling the number of cores yields only 20% to 40% increase in performance. This is a consecuence of NUMA, where the second socket is accessing has to perform remote accesses to the memory connected to the first socket throught the coherence network. In further experiments (not shown in this work), we were able to improve ThunderX performance using both sockets by introducing NUMA-aware directives in the code.

Network bandwidth comparison

In this section we present our experiments to obtain the peak bandwidth performance of our platforms.

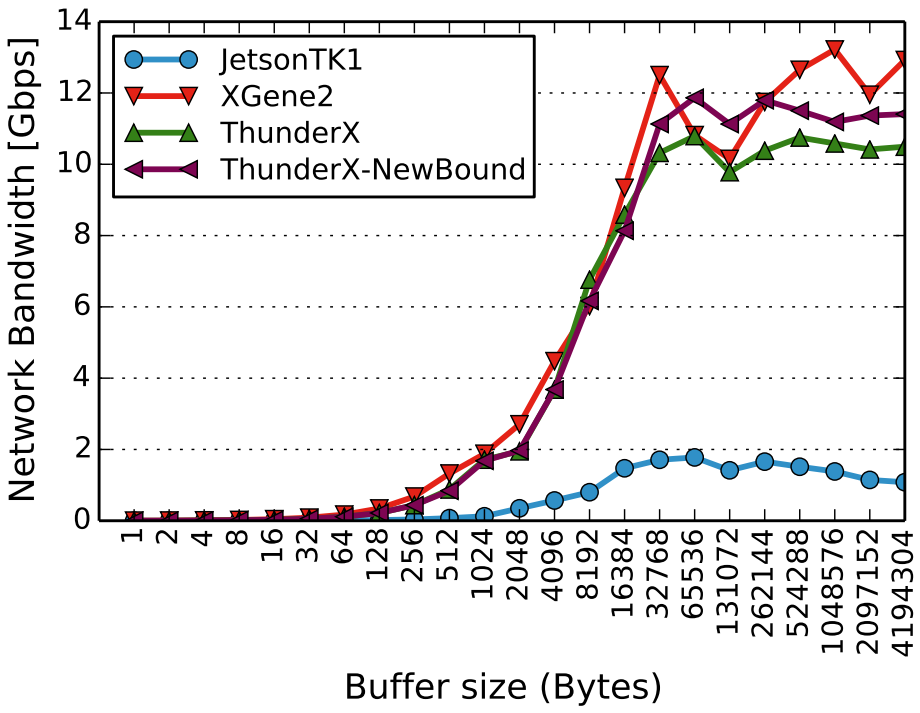


Figure 4.16: Network Bandwidth peak performance comparison scaling the buffer size.

Figure 4.16 shows the results of our network bandwidth stress test benchmark, multi-PingPong. As we mention in chapter 2, JetsonTK1, XGene and ThunderX platforms use 1 Gigabit ethernet, 10 Gigabit ethernet (with ROCE capabilities) and double aggregated link 10 Gigabit ethernet respectively. In the legend, *ThunderX* refers to the first interconnection configuration, where nodes are interconnected using one single 10 Gigabit ethernet link. *ThunderX-NewBownd* represents the performance with the later configuration, double 10 Gigabit ethernet aggregated link.

Between 16KB and 32KB we reach a plateau in performance, after that, performance decreases consistently in all plaftorms. Decrease in perfomance is caused by a change in the way MPI messages are sent. MPI implementations usually use two different techniques [17]. To send short messages (32KB and below) it uses an *eager* algorithm, where data is sent as soon as possible. To send long messages (above 32KB) it uses a rendezvous algorithm, where the sender sends a message asking for permision to send a bigger message, and waits for a response that confirms the

receiver is ready. In *rendezvouz* mode we are able to reach similar performance in all 64-bit platforms.

ThunderX obtains a 10% to 20% improvement in network performance from link aggregation. Note that, for this type of workloads, link aggregation is not supposed to yield great improvements.

4.3.3 Layer 2 benchmarks results

This section gathers our results and experiences with parallel mini-apps and benchmarks whose workloads have scientific relevance. First, we present our experiment results executing three *OpenMP-only* CORAL benchmarks: GFMCmk, MILCmk and AMGmk. And second, we present our results running *MPI-only* applications: HPCG, CoMD, LULESH and miniFE. At the end of this section, we share our observations and lessons learnt during the testing process.

Single node: OpenMP benchmarks

As we mentioned (see section 1.3.1), memory bandwidth is a critical resource in multicore processors. Previous section showed that even having similar floating point performance per core, memory bandwidth available per core could be different in orders of magnitude. In HPC workloads, memory contention has a high impact in parallel performance.

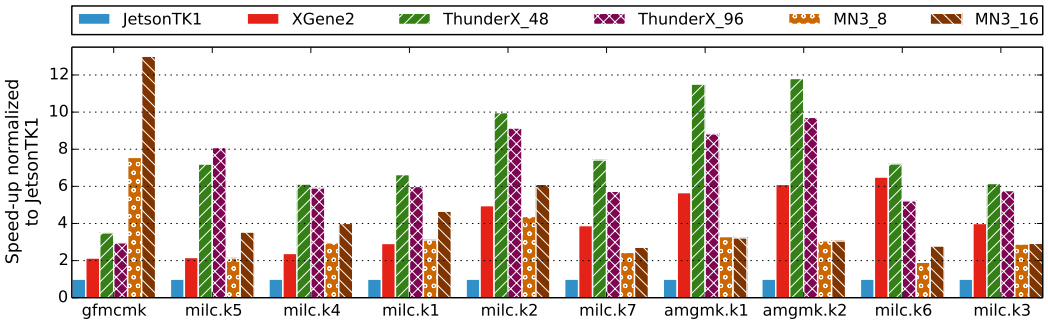
To understand better the behavior and results we present, we extract shared memory bus events statistics i.e: last level cache misses, writebacks and TLB misses; executing the different benchmark kernels. Table 4.1 shows the memory pressure statistics we obtained running each kernel in a single ThunderX socket (48 cores) ordered by L2 Cache Misses. Note that, these stats may differ from platform to

Benchmark	L2 MPKI	L2 WBPKI	TLB MPKI
gfmcmk	0.15	0.15	0.034
milc.k5	3.72	1.44	0.0048
milc.k4	5.05	2.19	0.006
milc.k1	6.22	1.33	0.0142
milc.k2	6.65	1.35	0.0133
milc.k7	7.666	0.0268	0.0232
amgmk.k1	11.31	0.96	0.034
amgmk.k2	11.15	2.05	0.0502
milc.k6	15.53	0.057	0.0454
milc.k3	28.23	16.17	0.1006

Table 4.1: Layer 2 OpenMP benchmarks memory stats.

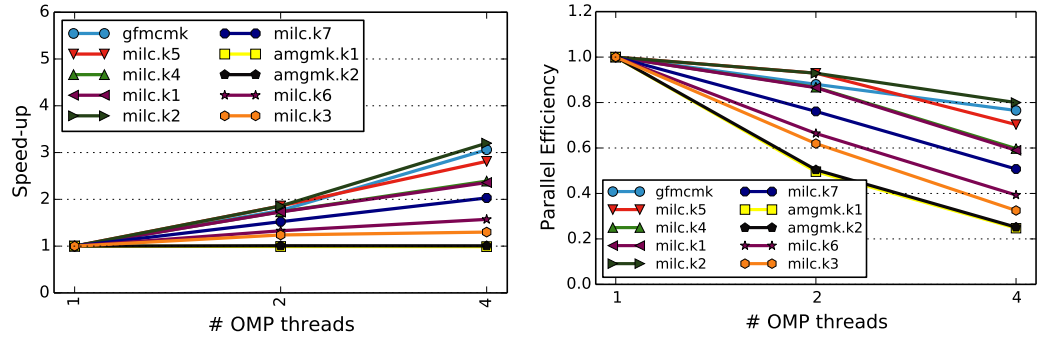
platform. However, we assume they are similar enough to do a coarse-grain discrimination between memory and compute intensive applications. All these metrics are obtained using Extrae and Paraver.

(a) Platform socket performance comparison.



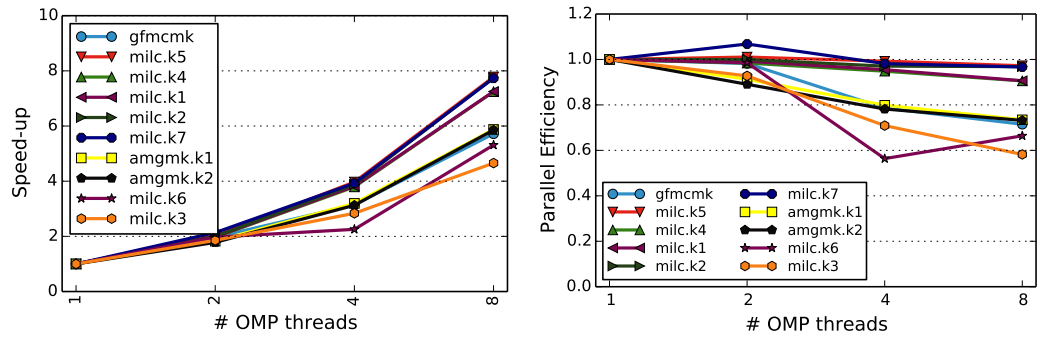
(b) Jetson strong scaling

(c) Jetson strong efficiency



(d) XGene2 strong scaling

(e) XGene2 strong efficiency



(f) ThunderX strong scaling

(g) ThunderX strong efficiency

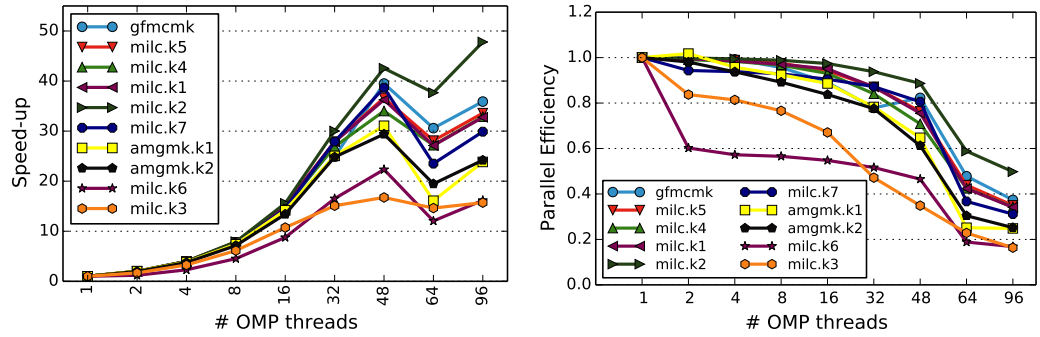


Figure 4.17: OpenMP comparison.

In figure 4.17 we gather all our results running OpenMP microkernel benchmarks. First, we compare Jetson, XGene2, ThunderX and MareNostrum3 performance using all cores in a socket (a). Second, for the three first platforms, we show paralel speedup and efficiency performing strong scaling tests (b, c, d, e, f, g). The order of appearance of the lines and bars in the plots is based on the L2 MPKI shown in the previous table; we leverage that ordering to reason about the memory bandwidth impact in terms of parallel performance.

About figure (a) we have several observations related to unexpected issues. First, as we see, in most of the benchmarks, ThunderX performance using one socket is higher than using two. We observe these critical performance issues when executing OpenMP kernels using both sockets in ThunderX (Thunder_96). We perform a detailed analysis of such issues in section 5.2.

By looking at the performance gap between platforms we observe that, in compute intensive kernels (gfmcmk, milc.k5, milc.k4) XGene2 is still 2x faster than JetsonTK1; but in memory intensive applications this gap increases up to 6x. Comparing ThunderX_48 to XGene2, as L2 Cache Misses increase (and so it does the memory bus contention), the performance gap is reduced in favor of the XGene2. Although we cannot reason coherently about ThunderX_48 and ThunderX_96, we observe that going up from 8 to 16 cores in MareNostrum3, i.e. one or two sockets, yields minimal performance improvements.

Figures (b, c) show JetsonTK1 strong scaling results. We observe that in general, as memory bus pressure increases, parallel efficiency drops very fast. Compared to other platforms JetsonTK1 obtain very bad scaling results even at 4 cores. These figures also suggest that AMGmk is experiencing additional issues, although we were unable to identify its source. Figures (d, e) show XGene2 strong scaling results. In terms of efficiency, at maximum number of cores on each platform (right furthest point), XGene2 obtains the best results in all applications having 90% efficiency for kernels with lower memory intensity and 60% for the most intensive ones. Additionally, if we compare XGene2 and ThunderX at 8 cores, ThunderX obtains better scaling results. We think that this is caused by the relative low throughput per

core of ThunderX, meaning that for the same number of cores in both platforms, the average memory bandwidth running the same benchmark in ThunderX is lower. Finally, in figures (f, g) aside from particular issues we mentioned before, at 48 cores ThunderX is efficiency on most applications is close to 75%, beyond that point efficiency drops down to a least 40% on most applications. In general in HPC, parallel efficiency below 75% is considered not acceptable.

Full cluster: MPI only benchmarks

In this section we present the results we obtain running MPI only applications. With this MPI benchmarks we are able to test the full architecture of the mini-clusters. We obtained all the results performing weak scaling; all benchmarks allow us to perform weak scaling but only a small subset offers strong scaling test support. Also, these applications have restrictions in the problem partitioning, and because of that, we are missing several *intermidiate* performance points in our plots.

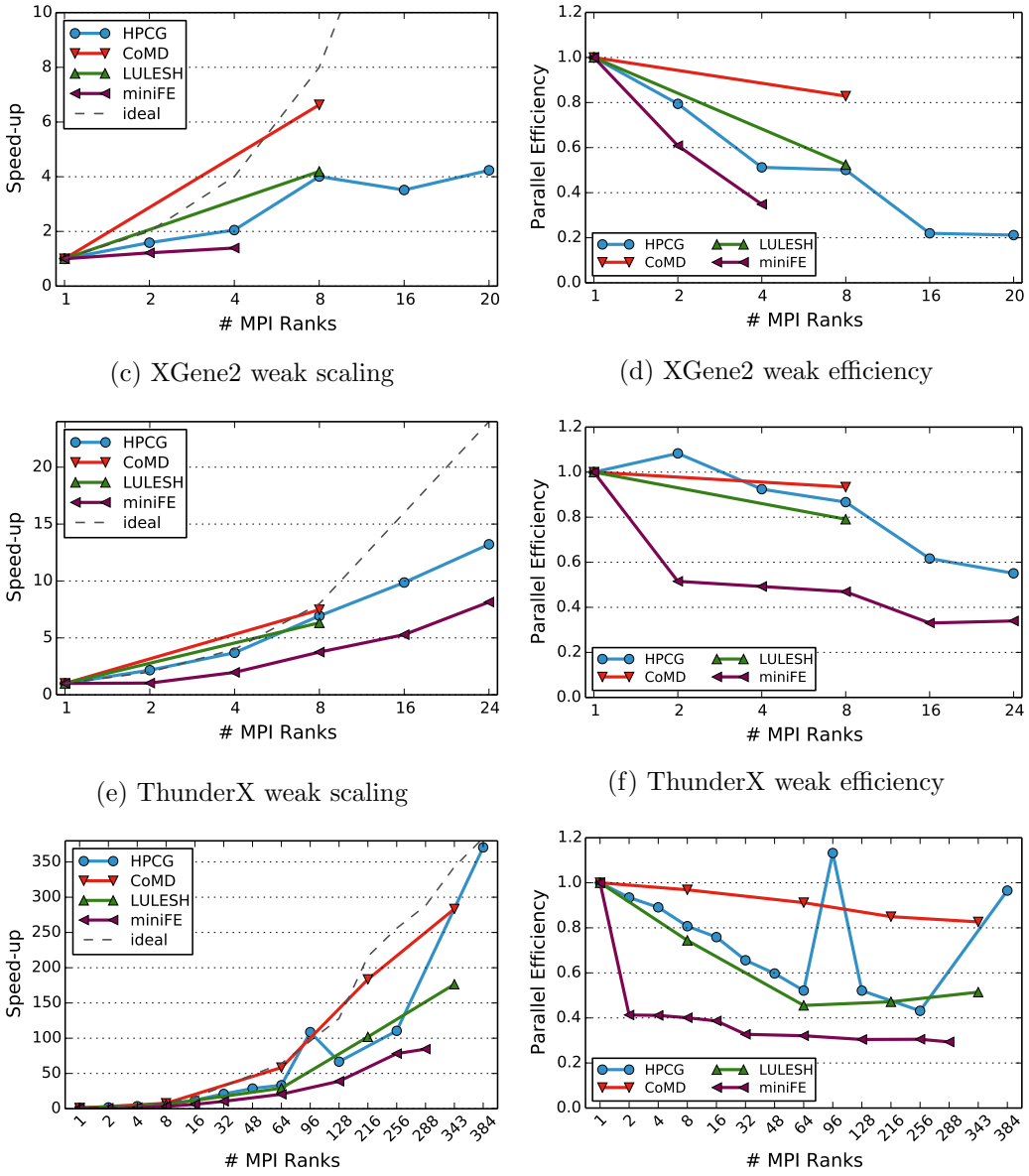
Figure 4.18, gathers our results running HPCG, CoMD, LULESH and miniFE. As in previous figures, we show parallel speedup and efficiency of each benchmark separated by platform. Although some of these applications support hybrid execution using OpenMP plus MPI, we disabled OpenMP in all benchmarks; in all tests, every process has it is own memory space.

In general, we observe that CoMD and LULESH are the more predictable applications in terms of performance. In the other hand, HPCG behavior differs heavily between platforms. miniFE suffers from big efficiency losses going up from one to two cores, but after that, scales reasonably well.

JetsonTK1 (see figures a,b), we only obtain reasonable scaling results running CoMD with 8 cores. The rest of results achieve poor scaling efficiency at 50% or lower.

(a) JetsonTK1 weak scaling

(b) Jetson weak efficiency



(c) XGene2 weak scaling

(d) XGene2 weak efficiency

(e) ThunderX weak scaling

(f) ThunderX weak efficiency

Figure 4.18: MPI comparison.

Comparing all platforms at 8 MPI Ranks, XGene2 obtains the best parallel efficiency; increasing the number of MPI Ranks inside a XGene2 (subfigures e, f) chip has minimal efficiency loss. However, we observe up to 25% efficiency losses when message passing over the network is required (executions with more than 8 cores). Our results show that XGene2 memory bandwidth allows us to scale applications reasonably well. Nonetheless, we should consider a deeper study of the network

performance running MPI applications to be able to optimize the network stack.

Finally, in subfigures (e) and (f) we observe that ThunderX parallel efficiency drops at a consistent rate in LULESH and HPCG for tests running 96 cores or lower. Over 96 cores, meaning the processes use the ethernet interconection for message passing, we observe sustained parallel efficiency at miniFE and LULESH and also a slower dropping rate in HPCG.

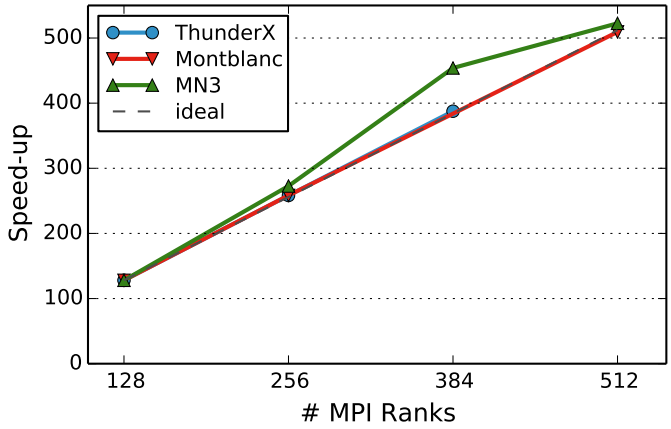
In our experience, in terms of flexibilty performing tests, these mini-app benchmarks are a step down compared to the previous OpenMP. The low number of nodes and diversity in number of cores in our platforms, increases the difficulty of performing fair comparisons. In other hand, the complexity of these applications also stepped up significantly. These benchmarks introduce more complex performance testing techniques (e.g: the several testing phases in HPCG), bigger codes and computational challenges like load unbalance. Testing with applications architecturally similar to the ones in HPC's state of the art provides relevant and useful insight but also requires an order of magnitude higher of dedication to understand the particularities of each benchmark or mini-app.

4.3.4 Layer 3 benchmarks results

In this section we present our results using production size scientific applications. As we mentioned, this testing layer is under development, and the only application we experiment with is Alya RED.

Figure 4.19, show scalability (a) and execution time (b) comparisons. We were only able to execute the application in ThunderX with a relevant input size for high number of cores; neither XGene2 or JetsonTK1 could allocate the problem in memory. In this case ThunderX scaling overlaps with the ideal scaling line. Note that, even executing with 384 cores, we are only using 4 ThunderX nodes, meaning

(a) Strong scaling



(b) Execution time comparison

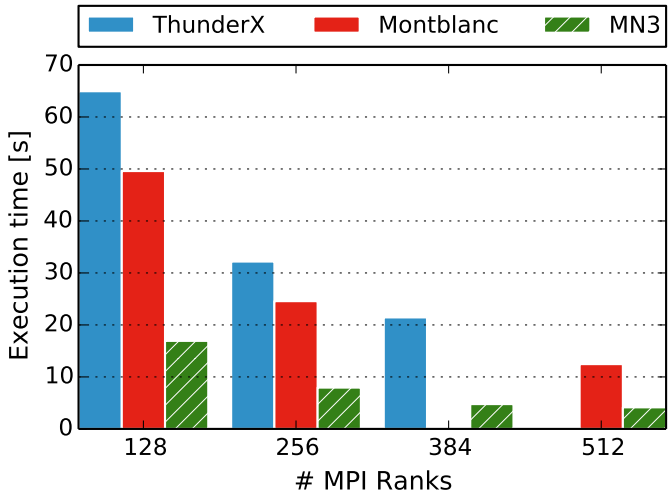


Figure 4.19: ThunderX results running Alya.

that the network has to handle a low number of remote connections. In terms of exection time, we observe how that using the same number of cores, ThunderX is 30% slower compared to Mont-Blanc.

Chapter 5

Development Issues

In this chapter, we describe some of the most important issues affecting the deployment of the prototype and mini-clusters. We also share our experiences and describe the actions that we performed, individually and as a team, in order to solve each issue. At the end, we also share our conclusions about the impact of these issues as a limiting factor to perform efficient HPC computing with low-power architectures.

To be more specific, the most common types of actions we needed to apply are: changes in the configuration of packages, upgrading user and system libraries, upgrading the linux kernel or apply hardware improvements. The majority of the issues we faced and solved during the development of a platform resulted in valuable lessons that helped us to deploy next prototypes faster.

5.1 Issues at the Mont-Blanc prototype

Here we reason about the major issues we encounter during the development of Mont-Blanc prototype. Later, we present our in depth study on one of such issues where that caused critical network performance issues.

5.1.1 Mont-Blanc prototype overall issues

Software stack maturity In section 4.1.2 and 4.1.3 we presented our experiences porting applications. Although, we had many setbacks during the deployment of the system software and libraries in Mont-Blanc prototype, we were able to succesfully develop a full HPC software stack. Having good communication between prototype users and sysadmins was one of the keys to this success. A important part of the state of the art libraries and compilers are designed with portability in mind, because of that, we were able to port them to ARMv7/8 only applying small changes in most cases. At the end, using our software stack we were able to provide support for complex applications like NMMB.

System unstability It is expected to deal with unstability during the development of a new platform. However, it has been a major source of slowdowns in our work. We use the term system unstability to refer to the issues, which we are not able to identify at the moment, that causes intermittent critical failures in the execution of our experiments.

File system Execution of jobs with a high number of nodes > 128 caused the Lustre filesystem to not sync files properly, even with low I/O activity; acceses to such files raised 'no such file or directory' errors. This is consequence of a bad design of the file server hardware configuration. Although it improved performance, upgrading to new version of Lustre both clients and servers was not enough to solve the issue. In our future cluster configurations we should review and and design a better I/O system that can handle high number of clients requesting file data and meta-data.

Network stack Three important issues affecting the bandwidth and latency of our network since the beginning were: the use of an ASIX driver optimized to work with USB1.0, wasting USB3.0 capabilities; the TCP retransmision timeout set at 200ms, causing huge delays when some packet is missing; and the use of only one 40 Gbps link at the top-of-the-rack switches. Upgrading the usb bridge driver, reducing retransmision time to 5ms and adding extra links up to 160 Gbps total bandwidth greatly improved the overall network performance.

Termal throttling We identified several nodes running permanently at low frequencies. It was affecting specially the nodes located at the *12th* node slot in the blades. This reveals a flaw in the cooling system and distribution of nodes in the board. We were forced to turn off those nodes, which also resulted in termal benefits to the rest of the nodes.

CPU frequency Instead of reporting the real operating frequency value, the operating system was reporting a fixed maximum. This issue was identified as a kernel bug. It was solved to regularly force low frequency values so the system adjusts the correct value itself.

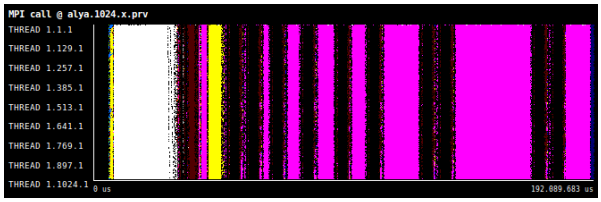
5.1.2 In depth study of very low MPI performance on Alya RED

As we mention in section 4.1.2 we experience critical performance issues in Alya RED when running with more than 1000 cores. We show the process we followed to properly identify the source.

Using extrae we obtained traces from executions with faulty iterations, which at the moment, were all of them. We study the traces using the paraver trace analysis tools. In the traces we located the faulty iterations and noticed irregular duration of MPI_Recv calls in the Master thread of the application. Such MPI_Recv occur during an all-to-one communication where all slave threads send computation results to the master thread. This messages, of about 1KB size, take hundreds of microseconds to complete in most cases, but in those faulty iteration they were taking from 5 to 20 seconds.

In figure 5.1a we quickly spot wider pink regions at the end of the trace, meaning that there is iterations taking way more time than others. Pink color represents MPI_Waitall synchronization barrier calls, meaning that threads will not progress until all the other threads reach that barrier. If we enable the communication lines and zoom in into one of these iterations (see figure 5.1b) we observe the all-to-one communication pattern we mentioned before. In this particular case, we are observing the first 71 (of 1023) slave threads, each one of them sending 2 messages of

(a) MPI Calls view



(b) Zoom into a faulty iteration

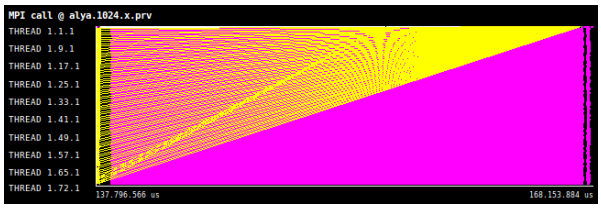
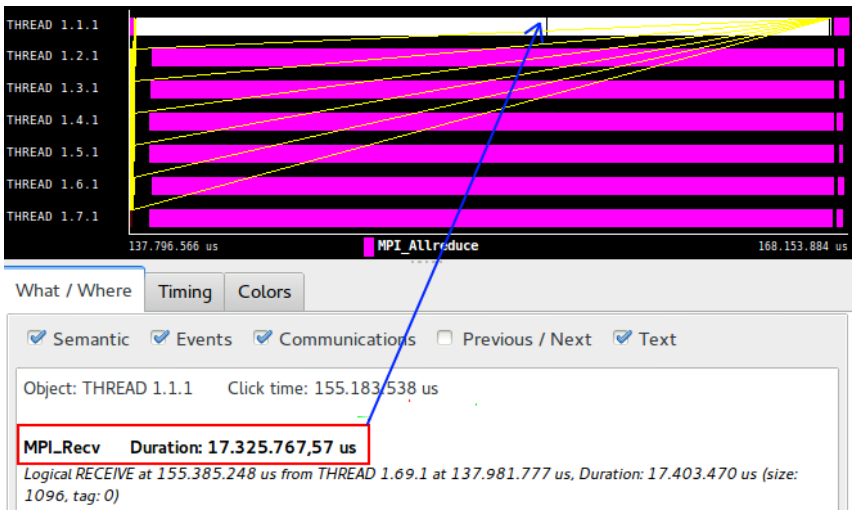


Figure 5.1: Trace showing irregular execution of Alya RED iterations on Mont-Blanc prototype.

around 1KByte.

Figure 5.2: Zoom into master thread during faulty iteration



In figure 5.2 we zoom again, now into the first 6 threads of the same faulty iteration (see 5.1b). We observe that a single MPI_Recv call (pointed with a blue arrow) takes 17 seconds to complete, being the size of that message exactly 1096 Bytes.

To discard that the source of this issue was the use of damaged compute nodes, we performed several runs shuffling arbitrarily the placement MPI Ranks in the physical nodes. To tackle this apparent we decide to take two approaches. First, traceback all the changes in the configuration of the network stack software. From this effort, the sysadmin team concluded that the change was Selective ACK in the configuration of linux implementation of tcp protocol. Second, install and test new linux kernel and network driver versions that included improvements for ARM platforms. This decision is made based on that other researchers also indentified several sources . The main changes this updates included were. Finally, enabling back the selective ack option allowed us to recover the initial performance levels, but as we see in the results of section 4.1.2, the additional improvements in the kernel and libraries allowed us to improve further in performance.

5.2 Issues at Cavium ThunderX

In our experience, deploying the full *Mont-Blanc* software stack in both ARMv8 platforms, APM XGene2 and Cavium ThunderX, required cosiderably less effort than on the Mont-Blanc prototype. Also, the relative low number of nodes in the mini-clusters leaves little room for relevant network issues. During our tests we did not encounter any relevant issue regarding the XGene2 platform. Alternatively, we observed recurrent and important performance problems in ThunderX. In this section we focus in these last problems.

Following, we present an in depth trace driven study of the critical performance issues we experience on ThunderX. Specifically, we detect huge workload unbalance and performance losses when running OpenMP applications using more than one socket. For this study, we obtained traces running the MILCmk and AMGmk OpenMP benchmarks with 48 (1 socket) and 64 cores.

5.2.1 MILCmk trace analysis

(a) milc.k1 (b) milc.k2

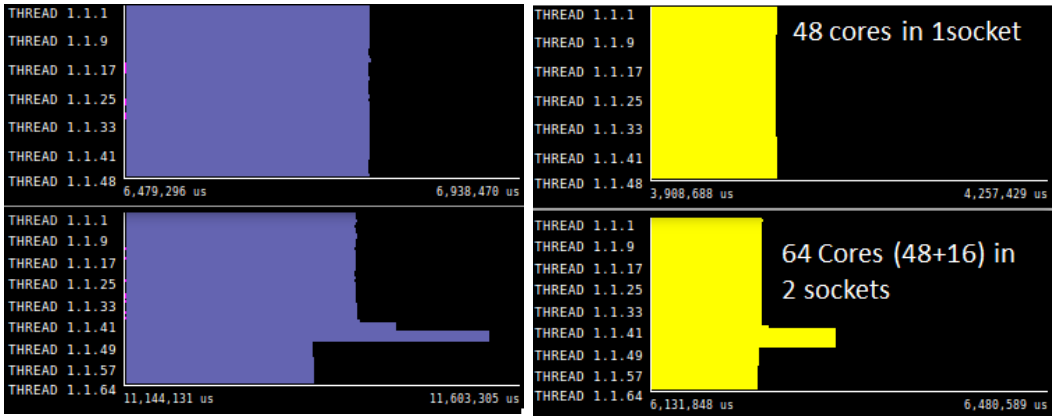


Figure 5.3: Trace showing irregular thread duration in two different MILCmk kernels.

In each figure 5.3a and 5.3b, we present Paraver timeline windows corresponding to the useful thread duration in a MILCmk kernel. In detail, in figure 5.3a we observe how a milc.k1 execution timeline with 48 cores (top) using only one ThunderX socket behaves coherently with the code of the kernel, an embarrasingly parallel workload. However, using 64 cores (bottom) we observe irregular duration on the range of threads 42 to 48. Note that top and bottom timeline views have the same timescale. Similarly, milc.k2 in figure 5.3b shows virtually the same behavior, again being the last threads of the first socket (threads 42 to 48) the ones experiencing duration unbalance. In this last kernel, slow threads duration averages 161ms, while regular thread duration averages 91ms, that is a 75% increase in execution time.

In our traces, we keep hardware counter information regarding the number of speculative instructions executed per thread and computational burst. Additionaly we keep information about other events like cpu cycles, memory stalled cycles, and so on. This allows us to obtain the specific values for each thread that correspond to the execution of one iteration of a milc kernel. We gather this information to study the possible correlation between those event values and the thread duration, and in consequence identify the source of the issue.

(a) Speculative instructions (b) Frequency (c) Cycles stalled in memory

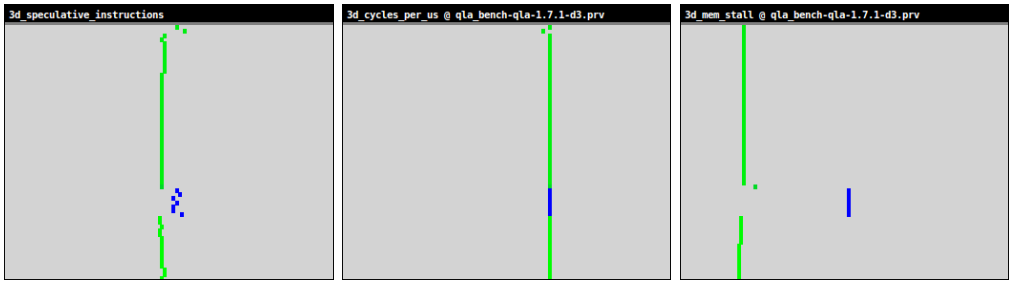


Figure 5.4: Histograms of one iteration of milc.k2 .

Figure 5.4 shows a set histograms corresponding to three metrics: speculative instructions, frequency and cycles stalled on the processor due to memory events. These metrics are obtained or derived from the hardware counter information. Along the Y axis of each histogram represents the OpenMP threads, meaning that any point colored in the first row represents an event value on thread 1, second row for the thread 2 and so on. Higher values of the metric are represented on the right side of the X axis. The points that appear in the histograms, correspond to the values we obtain during one iteration of the milc.k2 kernel, the same timeframe we show in the previous figures (see 5.3b). The coloring of each point is used to represent the time duration of them. Light green represents lower values while dark blue represents higher values.

In this case, the points colored in dark blue belong to the *problematic* (slower) threads. From there we observe three things. First, slower threads execute 0.05% more instructions compared to the rest. Considering that slower threads are 80% longer, a minimal increase in the number of instructions executed does not explain this behavior. Based on that, we discard load unbalance as a possible source of the issue. Second, each threads is running at the same frequency discarding faulty cores or preemptions. Finally, we observe an 2.3x increasing in the number of cycles stalled in memory for slower threads. Dividing the values by the frequency we obtain how much time a core is stalled waiting for memory instructions to complete, the result of that division is 65ms. Although, with this information we still cannot identify the specific issue, we have a strong indicator that it has to do with memory acceses.

This is a first study based on traces and hardware counters. Using this information we obtain more traces with other hardware counters, but now, all of them related to memory hierarchy events.

Further analysis of memory related hardware counters like L1 and L2 Data Cache Misses, show similar values as in executed instruction counters, where slow threads have $< 0.1\%$ increase in cache misses. Therefore, cache misses are not either the direct cause of the memory stalls.

5.2.2 AMGmk trace analysis

Before digging further in the MILCmk analysis we started performing the same tests with AMGmk. As we see in the following figures, we experience similar issues as in MILCmk.

(a) Timeline view (b) Histogram view

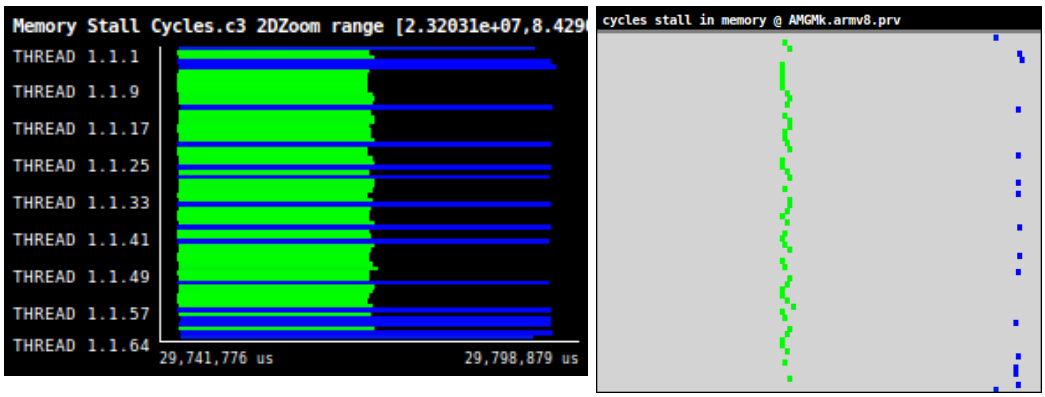
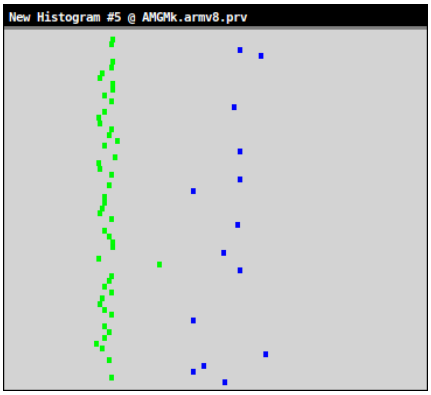


Figure 5.5: Trace showing irregular thread duration on AMGmk.

In our traces (see figure 5.5) we observe again direct correlation between cycles stalled in memory and extended execution time. And also (see figure 5.6), a very specific but minimal increase of number of memory instructions executed (below 0.01%) for those threads that take, in this case, almost double execution time for

the same task.

(a) Histogram view



(b) Histogram statistics

Total	261,170,854
Average	4,212,433.1290
Maximum	4,222,080
Minimum	4,209,518
StDev	3,890.0319
Avg/Max	0.9977

Figure 5.6: AMGmk memory instructions count histogram and statistics.

Finally, based on the data collected, we were not able to determine the exact source of this issue. Anyway, we think it is most likely an issue related to the mapping of virtual memory into physical memory causing that often threads need to access remote data. Our proposal to keep working solving the issue requires two actions. First, as a future work expand this trace driven study to analyze prefetcher and NUMA accesses during execution. And second, present this analysis to the Cavium tech support and get relevant information that allow us to identify and possibly fix this issue.

Chapter 6

Conclusions

In this chapter we share our conclusions and remark the main contributions of our work to the development of the Mont-Blanc project, along with observations about future improvements to extend our study.

We succesfully ported and evaluated three out of four scientific applications first on the Mont-Blanc prototype and later in our ARMv8 mini-clusters. On top of that, we have been actively involved to the development of the Mont-Blanc prototype software stack, contributing to two of the Mont-Blanc project main objectives: first, demonstrating the feasibility of running full production applications in an ARM based prototype, and also, improving the software ecosystem and paving the way for future ARM HPC systems.

Using our testing methodology we are able to compare several mini-clusters with different scopes. We discovered that the floating point performance per core of market available ARMv8 platforms slightly improves compared to ARMv7 platforms. Platforms with low memory bandwidth per core ratio like ThunderX obtain poor scaling results in common HPC OpenMP benchmarks that generate moderate activity in the memory bus. In our opinion, none of the ARMv8 plaftorms we tested is still mature enough to be used to build a HPC system. We noticed however that 64-bit platforms developed for sever market, offers better stability and better software support for running large productions codes, mainly due to the availability of

larger amout of memory per core.

In this thesis we described most of the important issues affecting the development of the prototype and mini-clusters. Also, we explain our approach in each case to solve the issues or the reason behind our failures. Based in our experiences, most of the HPC applications are designed to work in 64-bit environments, using 32-bit processors may imply several critical limitations and a non negligible effort by the developers to support this architecture. Compared to ARMv7, ARMv8 processors are a huge step forward in terms of compatibility with applications and libraries.

We would like to add some comments about some aspects of our work that we should consider for future work.

Except for Alya RED, we are missing energy consumption results of the applications. As we pointed out, this is caused by problems in the PMU across platforms and in some cases the lack of support for energy measurements.

Although we are satisfied with the information obtained based on the results using our testing methodology, we identified several aspects that require improvements. First, we can not ensure that our selected benchmarks represent the total spectrum of state of the art HPC applications, we need to perform a better study and classification of the computational workload they represent. We should also expand the set of applications on layer 3.

Constant setbacks and issues in the Mont-Blanc prototype did not leave room for developing specific optimizations in Alya RED or NMMB as it was one of our objectives in the project. Developing such optimizations require a deep understading of the application. Instead we invested our efforts in ensuring a correct porting and stable execution of applications.

To conclude, compared to the MareNostrum3 supercomputer, the Mont-Blanc prototype is ≈ 4 times slower running HPC applications with the same number of cores, while at the same time, obtains good parallel efficiency and similar energy efficiency.

Our results running Alya RED have been included in a paper that presents the architecture and performance results of the Mont-Blanc prototype. This paper has been accepted at the Supercomputing Conference 2016 that will be held in Salt Lake City during November 2016.

Acknowledgements

This research has been supported by the Mont-Blanc project (European Community's Seventh Framework Pro-gramme [FP7/2007-2013] under grant agreement n. 288777 and 610402), the Spanish Ministry of Science and Technology through Computacion de Altas Prestaciones (CICYT) VI (TIN2012-34557), the Spanish Government through Programa Severo Ochoa (SEV-2011-0067)

Bibliography

[1] David Abdurachmanov et al. “Heterogeneous high throughput scientific computing with apm x-gene and intel xeon phi”. In: *Journal of Physics: Conference Series*. Vol. 608. IOP Publishing, 2015, p. 012033. URL: http://iopscience.iop.org/article/10.1088/1742-6596/608/1/012033/meta (visited on 04/27/2016) (cit. on p. 22).

[2] David Abdurachmanov et al. “Initial explorations of ARM processors for scientific computing”. In: *Journal of Physics: Conference Series* 523 (June 6, 2014), p. 012009. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/523/1/012009. URL: http://stacks.iop.org/1742-6596/523/i=1/a=012009?key=crossref.cbfc6628245aff25d87335f5a9e08aa1 (visited on 04/27/2016) (cit. on p. 22).

[3] George Almási et al. “Optimization of MPI collective communication on BlueGene/L systems”. In: *Proceedings of the 19th annual international conference on Supercomputing*. ACM. 2005, pp. 253–262 (cit. on p. 18).

[4] hpcg benchmark.org. *HPCG homepage*. 2015. URL: http://www.hpcg-benchmark.org/index.html (cit. on p. 42).

[5] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. “Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures”. In: *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE. 2013, pp. 1–12 (cit. on p. 19).

[6] Barcelona Supercomputing Center. *Alya Homepage*. 2016. URL: http://bsccase02.bsc.es/alya/overview/ (cit. on p. 35).

[7] Barcelona Supercomputing Center. *NMMB-CTM Homepage*. 2016. URL: https://www.bsc.es/earth-sciences/nmmbbsc-project (cit. on p. 35).

[8] Barcelona Supercomputing Center. *SMUFIN Homepage*. 2015. URL: http://cg.bsc.es/smufin/ (cit. on pp. 35, 107).

[9] Nvidia Corporation. *JetsonTK1 Homepage and characteristics*. 2016. URL: http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html (cit. on p. 34).

[10] Toshio Endo, Akira Nukada, and Satoshi Matsuoka. "TSUBAME-KFC: A modern liquid submersion cooling prototype towards exascale becoming the greenest supercomputer in the world". In: *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2014, pp. 360–367 (cit. on p. 19).

[11] Michael A Heroux, Jack Dongarra, and Piotr Luszczek. "HPCG technical specification". In: *Sandia report SAND2013-8752* (2013) (cit. on p. 42).

[12] Oriol Jorba et al. "The NMMB/BSC-CTM: A multiscale online chemical weather prediction system". In: *HARMO 14: Proceedings of the 14th International Conference on Harmonisation Within Atmospheric Dispersion Modelling for Regulatory Purposes*. Vol. 14. 2011, pp. 345–349 (cit. on p. 37).

[13] Ian Karlin et al. "Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application". In: *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*. Boston, USA, May 2013 (cit. on p. 41).

[14] LLNL.gov. *Coral Benchmark Codes*. 2014. URL: https://asc.llnl.gov/CORAL-benchmarks/ (cit. on pp. 41, 67).

[15] LLNL.gov. *LULESH characteristics summary*. 2016. URL: https://asc.llnl.gov/CORAL-benchmarks/Summaries/LULESH_Summary_v1.pdf (cit. on p. 41).

[16] LLNL.gov. *LULESH Homepage*. 2016. URL: https://codesign.llnl.gov/lulesh.php (cit. on p. 41).

[17] LLNL.org. *MPI Performance Topics*. 2014. URL: https://computing.llnl.gov/tutorials/mpi_performance/ (cit. on p. 72).

[18] Mantevo.gov. *Mantevo Project Homepage*. 2016. URL: https://mantevo.org/ (cit. on pp. 42, 67).

[19] Satoshi Matsuoka. "Japanese HPC, Network, Cloud and Big Data Ecosystem circa 2015 onto Post-Moore". BDEC. 2015. URL: http://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/6-BDEC2015-Matsuoka-Japan-update-final.pdf (cit. on p. 17).

[20] Valentí Moncunill et al. "Comprehensive characterization of complex structural variations in cancer by directly comparing genome sequence reads". In: *Nature biotechnology* 32.11 (2014), pp. 1106–1112 (cit. on p. 39).

[21] Onur Mutlu. "Memory Scaling: A Systems Architecture Perspective". MemCon. 2013. URL: https://users.ece.cmu.edu/~omutlu/pub/mutlu_memory-scaling_memcon13_talk.pdf (cit. on p. 17).

[22] nasa.gov. *Nasa Parallel Benchmarks*. 2016. URL: http://www.nas.nasa.gov/publications/npb.html (cit. on p. 67).

[23] ncsa.illinois.edu. *Alya code scaled to 100,000 cores on Blue Waters supercomputer*. 2014. URL: http://www.ncsa.illinois.edu/news/story/alya_code_scaled_to_100000_cores_on_blue_waters_supercomputer (cit. on p. 36).

[24] princeton.edu. *The Parsec Benchmark Suite*. 2016 (cit. on p. 67).

[25] Nikola Rajovic et al. "Supercomputing with commodity CPUs: are mobile SoCs ready for HPC?" In: ACM Press, 2013, pp. 1–12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503281. URL: http://dl.acm.org/citation.cfm?doid=2503210.2503281 (visited on 04/14/2016) (cit. on pp. 13, 20, 22).

[26] Nikola Rajovic et al. "Tibidabo: Making the case for an ARM-based HPC system". In: *Future Generation Computer Systems* 36 (July 2014), pp. 322–334. ISSN: 0167739X. DOI: 10.1016/j.future.2013.07.013. URL: http://linkinghub.elsevier.com/retrieve/pii/S0167739X13001581 (visited on 04/14/2016) (cit. on p. 22).

[27] John Shalf. *The Exascale Challenge: How Technology Disruptions Fundamentally Change Programming Systems*. 2013 (cit. on pp. 16, 17).

[28] ohio state.edu. *Polybench/C Homepage*. 2016. URL: http://web.cse.ohio-state.edu/~pouchet/software/polybench/ (cit. on p. 67).

[29] Programming Models Team. *Mercurium Compiler Homepage*. 2016. URL: https://pm.bsc.es/mcxx (cit. on p. 40).

[30] Programming Models team. *Mercurium ticket tracker website*. 2016. URL: https://pm.bsc.es/projects/mcxx/ticket/2168 (cit. on p. 48).

[31] top500.org. *Performance Development — TOP500 Supercomputer Sites*. 2016. URL: http://top500.org/statistics/perfdevel/ (cit. on p. 16).

[32] top500.org. *Top 500 June 2016*. 2016. URL: http://www.top500.org/lists/2016/06/ (cit. on p. 23).

[33] ucar.edu. *[Bug 926219] New: netcdf: Does not support aarch64*. 2013. URL: http://www.unidata.ucar.edu/support/help/MailArchives/netcdf/msg11728.html (cit. on pp. 50, 103).

[34] J. S. Vetter and S. Mittal. "Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing". In: *Computing in Science Engineering* 17.2 (2015), pp. 73–82. ISSN: 1521-9615. DOI: 10.1109/MCSE.2015.4 (cit. on pp. 17, 19).

[35] University of Virginia. *STREAM benchmark Homepage*. 2016. URL: https://www.cs.virginia.edu/stream/ (cit. on p. 41).

[36] virginia.edu. *Rodinia Benchmark Suite*. 2016. URL: https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:A_Benchmark_Suite_For_Heterogeneous_Computing (cit. on p. 67).

[37] Wikipedia.org. *Amdahl's law*. 2016. URL: https://en.wikipedia.org/wiki/Amdahl%27s_law (cit. on p. 17).

Annex

A Installation guides

In this section we share the prototype and mini-clusters installation guides of all Mont-Blanc Severo Ochoa applications we worked with.

A.1 Alya RED installation guide

Dependences

- metis 4.0 Library for partitioning of finite element meshes. It is provided with Alya in Thirdparties/metis-4.0

Configure and Build

Starting at the root path of Alya RED folder

Step 1: Configure

This are the configure parameters required to work with gfortran

```
f90==    mpif90 -O1 -J$O -I$O -c -ffree-line-length-none
f77==    mpif90 -O1 -J$O -I$O -c
fpp90==  mpif90 -O1 -J$O -I$O -c -cpp -ffree-line-length-none
fomp90==  mpif90 -O1 -J$O -I$O -c -cpp -ffree-line-length-none
fpp77==  mpif90 -O1 -J$O -I$O -c -cpp -ffree-line-length-none
#cpp==    mpicc -no-multibyte-chars -c
cpp==    mpicc -c
link==   mpif90 -O1
```

```
libs==   -L../../Thirdparties/metis-4.0 -lmetis
fa2p== mpif90 -J../../Utils/user/alya2pos -I../../Utils/user/alya2pos -c
    -cpp
```

```
# Syntax: ./configure -x -f=<your_config_file> [ module 1 | module 2 | ...
    ]
# Alya RED configure command
$> ./configure -x -f=configure_MB_ARMv7l_mpif90_gfortran.txt parall exmedi
    solidz
# Step 2, Build
# After executing configure command, build with:
$> make
#Run
#Change directory to the folder containing your Alya RED input data.
#$> mpirun -np 3 /path/to/Alya.x <Test Name>
```

A.2 NMMB-CTM installation guide

File and Folder structure

We assume pathtoNMMB_vX.Y.Z as root folder for the install instructions.

DATA Contains the STATIC Database and the INPUT samples for NMMB.

MODEL Build directory for NMMB.

LIBS Build directory for the libraries needed for NMMB

SRC NMMB Source files

SRC_LIBS NMMB needed libraries source files

Configure and Build

First of all, we need to build the following **dependencies** provided inside the package:

- ESMF
- bacio
- Makedepf90
- nemsio
- cnvgnib
- netcdf
- sigio
- sp
- w3
- wgrib

AARM64 additional fixes

netCDF autoreconf At netCDF source folder: autoreconf -f -i. The problem and its solution were posted in the official forums [33]

Edit ESMC_Config.h In ESMC float sizes are specified by DEFINE clause that is set depending on the target architecture. By default, AARM64 flag is not supported. We added a line forcing them to be 8 (8bytes)

makefim KSH (KornShell) is a dependence to make and make clean NMMB. Can be easily avoided by editing the file that call it and doing this changing the *makefim clean* command to simple *make clean*.

Building dependencies

ESMF

Followin environment variables are required for building. We create a file *esmf-env-vars.sh* that includes this vars:

```
#!/bin/sh
export ESMF_OS=Linux
export ESMF_DIR=/path/to/nmmb/MODEL/SRC_LIBS/esmf_6_3_0r/
export ESMF_INSTALL_PREFIX=/path/to/nmmb/MODEL/SRC_LIBS/esmf_6_3_0r/
export ESMF_COMM=mpich2
export ESMF_BOPT=O
export ESMF_OPTLEVEL=3
export ESMF_ABI=32
export ESMF_COMPILER=gfortran
export ESMF_SITE=default
\end{lstlistings}

# Then execute the script and build with make

source esmf-env-vars.sh
make
```

Bacio

For Mont-Blanc prototype configure *makebacio.sh* as follows:

```
#    Update 4-byte version of libbacio_4.a
export LIB="../../libs/libbacio_4.a"
export INC="clib4.h"
export FFLAGS="-O3"
export AFLAGS=""
export CFLAGS="-O3 -mcpu=cortex-a15 -mtune=cortex-a15 -mfloat-abi=hard
    -mfpu=vfpv3-d16 "
make -f make.bacio

#    Update 8-byte version of libbacio_8.a
 export LIB="../../libs/libbacio_8.a"
 export INC="clib8.h"
 export FFLAGS="-O3 -fdefault-real-8 -fdefault-integer-8 "
 export AFLAGS=""
 export CFLAGS="-O3 -mcpu=cortex-a15 -mtune=cortex-a15 -mfloat-abi=hard
```

```
    -mfpu=vfpv3-d16 "
 make -f make.bacio
```

Original ifortran config:

```
#    Update 4-byte version of libbacio_4.a
#
export LIB="../../libs/libbacio_4.a"
export INC="clib4.h"
export FFLAGS="-O3 -fp-model precise"
export AFLAGS=""
export CFLAGS="-O3"
make -f make.bacio
#
#    Update 8-byte version of libbacio_8.a
#
 export LIB="../../libs/libbacio_8.a"
 export INC="clib8.h"
 export FFLAGS="-O3 -i8 -r8 -fp-model precise"
 export AFLAGS=""
 export CFLAGS="-O3 -m64 "
 make -f make.bacio
```

makedepf90

```
CFLAGS="-O3 -mcpu=cortex-a15 -mtune=cortex-a15 -mfloat-abi=hard
    -mfpu=vfpv3-d16" && ./configure
make
```

Installation notes: We had problems building with make because we did not have bison installed. For some reason lexer.c was empty and was causing problems.

nemsio

1. Edit conf/configure file

```
SHELL      = /bin/sh
```

```
FC          = mpif90
FREE        = -ffree-form
FIXED       = -ffixed-form
FFLAGS      = -O3
CC          = gcc
CCFLAGS     = -DLINUX -mcpu=cortex-a15 -mtune=cortex-a15 -mfloat-abi=hard
    -mfpu=vfpv3-d16
AR          = ar
ARFLAGS     = -rvu
RM          = rm
```

Original file was configured as follows:

```
\[...\]
CC          = icc
FREE        = -free
FIXED       = -fixed
\[...\]
```

netCDF

In command line:

```
$ > ./configure --prefix=path/to/nmmb-libs
    make
    make check
    make install
```

sigio

We edited makefile_4 as follows:

```
LIB    = ../../libs/libsigio_4.a
INCMOD = ../../libs/incmod/sigio_4/
FC  =  gfortran
FFLAGS = -ffree-form -O3 -I$(INCMOD)
```

```
FFLAGB = -ffree-form -ffixed-form -O3
AR  = ar
ARFLAGS = -ruv
```

A.3 SMuFiN installation guide

Getting the source code and input files

Development team provided a temporary SVN repository to obtain the latest version of the code. The one available at genomics group homepage is out of date [8].

```
svn co https://svn.bsc.es/repos/smufin
```

Input files can be obtained from *http://cg.bsc.es/smufin/ ¿ Downloads ¿ Example dataset*. Dataset is called ch22_insilico, and contains the sequences for the chromosome 22 from a healthy cell and a tumoral cell. Size uncompressed is around 13G.

Installation at MN3

Compiler: g++ (GCC 4.9.1); MPI: OpenMPI 1.8.1

For better performance input files (13GB) should be placed at: /gpfs/scratch/ and source code and bin at: /gpfs/projects/

Load GCC 4.9.1 and OpenMPI 1.8.1 modules

```
$> cd /path/to/sources/trunk
$> make
```

BSUB file example configured to obtain traces with EXTRAE

```
#!/bin/bash
#BSUB -J 32t_ch22_insilico
#BSUB -n 32
#BSUB -o /path/to/smufin/32t_ch22_insilico/32t_ch22_insilico_%J.out
#BSUB -e /path/to/smufin/32t_ch22_insilico/32t_ch22_insilico_%J.err
#BSUB -cwd /path/to/smufin/trunk/
```

```
#BSUB -W 04:00
#BSUB -x
#BSUB -R "span[ptile=4]"

module purge
module load gcc/4.9.1
module load openmpi
module load EXTRAE

mpirun trace_mn3.sh ./SMuFin --ref
    /gpfs/scratch/bsc18/bsc18880/dataset/ref_genome/hg19.fa
    --normal_fastq_1
    /gpfs/scratch/bsc18/bsc18880/dataset/normal_fastqs_1.txt
    --normal_fastq_2
    /gpfs/scratch/bsc18/bsc18880/dataset/normal_fastqs_2.txt
    --tumor_fastq_1 /gpfs/scratch/bsc18/bsc18880/dataset/tumor_fastqs_1.txt
    --tumor_fastq_2 /gpfs/scratch/bsc18/bsc18880/dataset/tumor_fastqs_2.txt
    --patient_id chr22_insilico --cpus_per_node 4
```

Note: trace_mn3.sh was modified in order to be able to solve the following error.

```
/.statelite/tmpfs/gpfs/home/bsc18/bsc18880/apps/smufin/trunk
--------------------------------------------------------------
mpirun noticed that process rank 22 with PID 4489 on node s16r1b49
exited on signal 11 (Segmentation fault).
```

Code added at the end of trace_mn3.sh:

```
if [ ! -z "${TMPDIR}" ]; then
    export TMPDIR=${TMPDIR}/extrae
    mkdir -p ${TMPDIR}
fi
```

A.4 Saiph installation guide

File structure

The basic file structure of OmpSs/CL applications generated from Saiph sources looks like this:

myApp.cpp which contains the C++ code w OmpSs support.

myApp.cl which contains the definition of the OpenCL kernels

saiph.h which contains some common headers to execute Saiph-generated apps

Configure and build

Dependences and environment variables:

- Mecurium compiler
- BOOST (ver. >= 1.42)
- VTK (ver. >= 6.1)

Load following modules environment:

- gcc/4.9.0
- opencl/1.1.0
- ompss/stable

B Compilation Flags

ifotran flag	**gfortran flag**	**Use**
-01	-01	Optimization level. Optimizations applied may differ between compilers.
-module	-J $0	Module path for libraries
-$O	-I $0	Include Path for objects
-c	-c	compile (no linking)
-fpp	-cpp	Enable preprocessing
-traceback	-g	Includes debug information in the binary file

Table 1: Ifortran to gfortran flag equivalences.